ECRTS 2014

9-11 July

Madrid, Spain

ECRTS 2014

26th Euromicro Conference on Real-Time Systems

Madrid, Spain
July 8 - 11, 2014

Work-in-Progress Proceedings

Edited by Marko Bertogna

Message from the Work-in-Progress Chair

Dear Colleagues:

Welcome to Madrid, and to the Work-in-Progress (WiP) Session of the 26th Euromicro Conference on Real-Time Systems (ECRTS'14). This session is dedicated to promising new and ongoing research on real-time systems and applications. I am happy to present seven WiP papers that cover innovative research from a spectrum of topics, including multicore scheduling and timing analysis, functionality-aware task scheduling, real-time network architecture, dependable and real-time wireless communications, and component-based design and model-based development. I am confident that many of the research contributions we feature here will appear as full-fledged conference and journal papers in the near future. The proceedings will be published online via the ECRTS 2014 WiP website: http://ecrts.eit.unikl.de/index.php?id=wip11 .

The primary purpose of the WiP session is to provide researchers with an opportunity to discuss their evolving ideas and to gather feedback from the real-time community at large. Due to time constraints, the presentations in this session can only provide a brief overview of the new creative ideas and interesting approaches of the selected research contributions. Nevertheless, I hope that you all will enjoy this session, and that you will find the ideas presented interesting. Most of all, I hope you will participate in stimulating discussions, exchange your ideas, and provide valuable feedback to the authors.

I would like to thank the members of the WiP session technical program committee for their help in reviewing the papers. I would also like to thank the authors for their interesting contributions and their confidence in ECRTS as a means to improve and advance their research. Last but not least, special thanks go to the ECRTS'14 organizers, Juan de la Puente, Rolf Ernst, and Gerhard Fohler, for their support.

Marko Bertogna
Real-Time and Algorithmic Research Group
University of Modena, Italy
ECRTS 2014 WiP Chair

ECRTS 2014 Work-in-Progress Technical Program Committee

Table of Contents

Guaranteeing Schedulability of Splittable Hard Real-Time Tasks for Non-Preemptable Devices

Mitra Nasri
Chair of Real-time Systems,
Technische Universität Kaiserslautern,
Germany
nasri@eit.uni-kl.de

Gerhard Fohler
Chair of Real-time Systems,
Technische Universität Kaiserslautern,
Germany
fohler@eit.uni-kl.de

Nafiseh Moti
School of Electrical & Computer Engineering,
University of Tehran, Tehran,
Iran
n.moti@ut.ac.ir

Abstract—**Nowadays it has been possible for many embedded systems to use peripheral computational resources such as graphics processing units (GPUs) to execute hard real-time data-parallel applications. However, because of the contentions on the buses and memory access channels, or the limited interface of the devices such as GPUs, the executing application must not be preempted while it is occupying the resource. Since non-preemptive scheduling of hard real-time periodic tasks is an NP-Hard problem, one efficient solution is to split the task into non-preemptive chunks. In this paper we introduce fine-grained periodic resource model (FG-PRM) which is customized for non-preemptable devices. Using the notion of reservation task model, we derive schedulablity condition as a function of the resource period, then we solve it by applying mathematical function estimation. The resulting parameter assignment method have been evaluated by its acceptance ratio. We have demonstrated the efficiency of our method through the experiments.**

I. Introduction

Nowadays the use of multi-core and many-core architectures as powerful computational resources has been increased in embedded and cyber-physical systems [1]. Using these resources it is possible to have hard real-time applications with significant computational requirements such as emergency collision avoidance in automatic cars and intelligent cruise control [2] running on the many-core peripheral computational devices such as graphics processing units (GPU). Many of these applications are data-parallel and they can be split into smaller chunks [3].

For many reasons such as contentions on the buses and memory access channels, or the use of GPUs which communicate with the processor through interrupt based interface, the executing application must not be preempted while it is occupying the resource [1], [4]. However, guaranteeing schedulability of a non-preemptive system with periodic tasks is an NP-Hard problem [5]. One efficient solution is to split the task into non-preemptive chunks. Many applications such as those consisting of matrix multiplications [3], [2] support flexible split sizes, however, it has to be performed at design time. One of the pioneer solutions to split such tasks for guaranteeing soft deadlines for non-preemptive accessible devices like GPUs has been introduced in [6], however, to the best of our knowledge, there is no solution to guarantee hard deadlines.

In this paper we use the notion of periodic resource model (PRM) [7] and make it customized for non-preemptable devices. PRM describes the resource behavior as a periodic model with two phase; the resource is available for duration C then it becomes idle. This pattern will be repeated periodically with period P. To determine high priority tasks to be executed during the execution budget within the active phase, PRM uses an ordinary scheduler such as RM which might result in task's preemptions because the length of active phase might be greater than period of some of the tasks. Moreover, the cost of exact schedulability analysis of this model is relatively high [8] and current approximated solutions such as [9] are not efficient for all task sets, specially when the ratio between the shortest and the longest period is large. In those cases, outputs of [9] and [8] will produce pairs $C \approx P$ while the actual required resource might be far less than the resulting parameters. Due to these drawbacks, the existing feasible parameter assignment methods for PRM are not computationally affordable or efficient for our target systems.

PRM can be considered as a special type of server based solutions which has one periodic task to handle all other tasks in the system. Although the traditional usage of these approaches was to effectively execute the aperiodic tasks [10], by the introduction of constant bandwidth servers [11], the temporal partitioning model [12], and the reservation task model [13], these methods have been extended to provide isolation for the tasks or to handle sporadic tasks as periodic ones [13]. However, because of the parameter assignment methods applied in the most of these studies, tasks might be preempted inside the execution budget of the server task or because of the preemptions caused by other periodic tasks.

In this paper, we introduce fine-grained periodic resource model (FG-PRM) as a customized PRM that guarantees non-preemptive execution inside the active phase. To construct the basic parameter assignment problem as a function of the PRM parameters, i.e., C and P, we use the formulations of the reservation task model [13] which assigns one server to one task. However, to construct our customized PRM, we represent the task set with only one reservation task. Thus, in every period, each task has one portion of execution with a fixed length, and the tasks finish their execution after a fixed number of releases of the reservation task. To solve the parameter assignment problem we use mathematical function approximation. In summary, in our method, the schedulability of each task is guaranteed because of the use of the reservation task model formulation, while non-preemptive execution is guaranteed by assigning fixed non-interleaving execution windows to the tasks.

Our solution, not only helps GPU applications but also provides a mean to feasibly schedule any other application on non-preemptable I/O devices as long as flexible splitting is permitted. It can be applied on periodic or sporadic tasks with arbitrary release offsets or explicit deadlines shorter than period. Besides, our approach does not need real-time scheduling algorithms; it provides a prioritized queue of the tasks with specified execution budget for each task within which the task can be executed non-preemptively. Size of these budgets (or splits) is obtained as a parameter of the model at design time, hence, it is available for the applications beforehand. Moreover, we are able to obtain the upper bound of the response time of the tasks and its error bound in $O(1)$.

The reminder of the paper is organized as follows; Sect. II introduces the system model. In Sect. III, the concept our work is presented which is followed by parameter assignment method in Sect. IV. Few experimental results have been presented in Sect. V and the paper is concluded in Sect. VI.

II. System Model

We consider a set of hard real-time independent sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task is identified by $\tau_i : (r_i, c_i, T_i, d_i)$, where r_i is the release offset, c_i is the execution time, T_i is the minimum inter-arrival time (period), and $d_i \leq T_i$ is the relative deadline of the task $\tau_i \in \tau$. The tasks have been indexed according to their period such that $T_i \leq T_{i+1}$, $1 \leq i < n$. Since we assume non-preemptable devices, c_i can be considered as resource occupation time, however, it has to be possible that we split c_i into smaller values at least at design time. The device is able to be used by each application for a specified amount of time, however, during this time, it cannot be preempted. GPUs are an example for such devices [2]. When a task enters GPUs, it will not leave it until end of its execution. Then GPU device notifies the operating system using an interrupt based interface. Although it is possible to split the tasks into smaller chunks and run those chunks on GPU device non-preemptively, size of the splits has to be known at compile time as mentioned in [6].

III. Fine-Grained Periodic Resource Model

Fine-grained periodic resource model governs the device as a PRM with period P and budget C. In the active phase, a queue of tasks is gradually sent to the device each of them has o_i units of device occupation time (DOT). The order of the tasks in the queue can be either fixed or dynamic, yet none of these options has any impact on the schedulability. For the simplicity we assume tasks will occupy the device according to their index from τ_1 to τ_n unless they are not released before their dedicated dispatch time. The later case happens when the original task is not in the system because of the assumptions regarding explicit deadlines and sporadic releases. If such tasks come later than their assigned window, they can be dispatched to the device otherwise their window will be simply ignored.

FG-PRM is based on reservation tasks [13] where for each original task, one reservation task is assigned. Theis and Fohler [13] have proven that if the parameters of the reservation task are selected in the following way, it can always guarantee schedulability of the original task as long

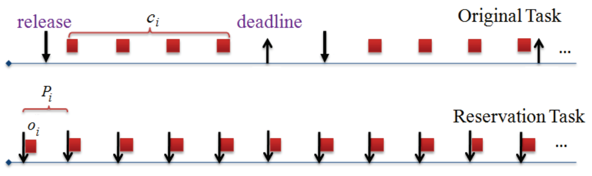

Fig. 1. An example of task execution using reservation task model [Theis11]

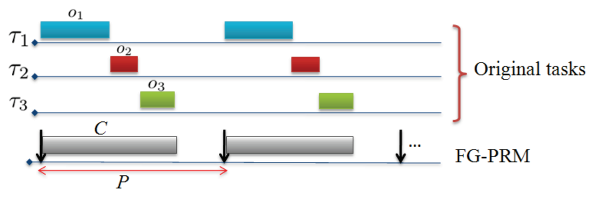

Fig. 2. An example of execution using FG-PRM

as the reservation tasks can be feasibly scheduled by some the underlying scheduling algorithm.

$$o_i = c_i / k_i \tag{1}$$

$$p_i = \frac{d_i + o_i}{k_i + 1} \tag{2}$$

where $k_i \in \mathbb{N}^+$ is the number of releases of the reservation task under one release of the original task, and p_i is the period of the reservation task. Fig. 1 shows the worst case scenario happened in the execution of the original task because the first release of the reservation task cannot be used to execute the original task. Besides, in our model (shown by Fig. 2), tasks are not allowed to be split at run time, hence, they can be executed only if they are released before the scheduled window of the reservation task. Consequently, we have to consider $k_i + 1$ releases with at least k_i effective releases.

If all tasks are scheduled by one reservation task, we can replace p_i by P in (2), then k_i and C are obtained as

$$k_i = \begin{cases} d_i/P & d_i/P \in \mathbb{N} \\ \lfloor d_i/P \rfloor - 1 & otherwise \end{cases} \tag{3}$$

$$C = \sum_{i=1}^{n} c_i / k_i \tag{4}$$

The resource utilization U^R is obtained by using the worst case value of (3) in (4) as

$$U^R = \frac{1}{P} \sum_{i=1}^{n} \frac{c_i}{k_i} = \frac{1}{P} \sum_{i=1}^{n} \frac{c_i}{\lfloor \frac{d_i}{P} \rfloor - 1} \tag{5}$$

Theorem 1. *Task set τ is schedulable by FG-PRM with period P and budget C if $U^R \leq 1$.*

Proof Having $U^R \leq 1$, all tasks have their dedicated time slot in every active phase. Also according to (1) each original task is guaranteed to have $k_i o_i = c_i$ DOT before its deadline, hence, it can be feasibly scheduled. ■

Fig. 3 shows U^R as a function of P for one task in two cases. According to this figure, U^R increases with the increase

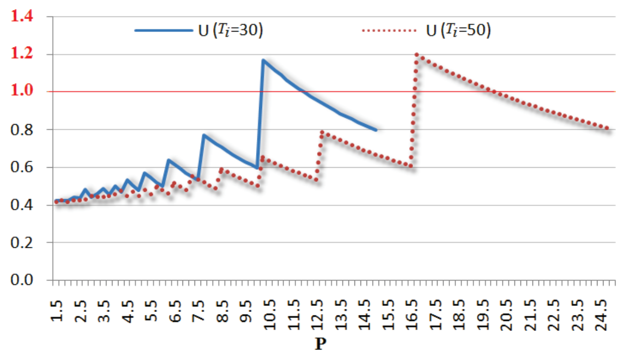

Fig. 3. U^R as a function of P for one task in two cases; $T_i = d_i = 30$ and $T_i = d_i = 50$ and $u_i = 40$

in P. Also it grows fast if tasks have small d_i. If P tends to 0, U^R tends to $U = \sum \frac{c_i}{d_i - 1}$ since

$$\lim_{P\to 0} \frac{1}{P} \sum_{i=1}^{n} \frac{c_i}{\lfloor \frac{d_i}{P} \rfloor - 1} = \lim_{P\to 0} \sum_{i=1}^{n} \frac{c_i}{P\lfloor \frac{d_i}{P} \rfloor - P} \le \sum_{i=1}^{n} \frac{c_i}{d_i - 1} \tag{6}$$

Since many of our target applications have considerable computational requirements, they usually have large periods as well. Yet it has to be mentioned that maximum value of P in our model is $0.5d_1$ and it happens when $k_1 = 2$. As shown by Fig. 1, response time of the tasks will be $WCRT_i = d_i$ and its error is

$$WCRT_i^{err} \le 2p_i - o_i \tag{7}$$

because in the best case scenario, the execution of the original task starts in the first release of the reservation task and finishes after o_i DOT after the release of the second-to-last reservation task. Both WCRT and its error are obtained in $O(1)$. Since WCRT of the task is equal to their deadlines, our model is efficient for the hard real-time applications which are not sensitive to long response times as long as deadlines are guaranteed. Moreover, in our work the overheads of the task splitting has been ignored. As a future work, we will apply [13] formulation to consider extra overheads of splitting into the utilization formulation. It is worth mentioning that our solution might not be efficient for dense task sets with tight deadlines and wide period values since it might have many ineffective reservations.

IV. Parameter Assignment Method

In this section we solve the parameter assignment problem. Upper and lower bounds of (5) can be obtained when $\lfloor \frac{d_i}{P} \rfloor$ is replaced by $\frac{d_i}{P} - 1$ and $\frac{d_i}{P}$ respectively. Thus we have

$$U^{up} = \frac{1}{P} \sum_{i=1}^{n} \frac{c_i}{\frac{d_i}{P} - 2} = \sum_{i=1}^{n} \frac{c_i}{d_i - 2P} \tag{8}$$

Fig. 4 illustrates the upper bound (8) for one task. Using Theorem 1 we obtain the schedulability problem as

$$U^{up} \le 1 \Rightarrow \sum_{i=1}^{n} \frac{c_i}{d_i - 2P} \le 1 \tag{9}$$

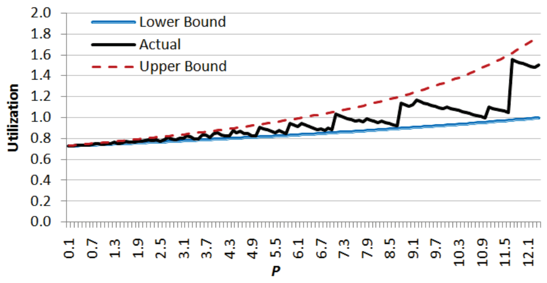

Fig. 4. U^{up}, U^R, and U^{low} as a function of P for a task set with 3 tasks $\tau_1 : (0, 12, 35, 35)$, $\tau_2 : (0, 10, 55, 55)$, and $\tau_3 : (0, 20, 99, 99)$.

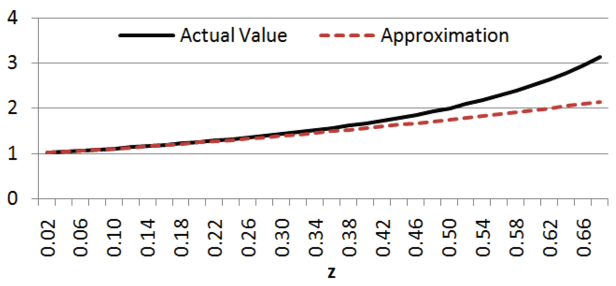

Fig. 5. The actual value of $\frac{1}{1-z}$ and its approximated value using the first 3 terms of (10).

However, it is not easy to solve (9) because the variable P appeared in the denominator of a sum. To make it solvable we use the following equality in Geometric series

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + ... \tag{10}$$

where $z < 1$. To keep our formulation solvable, we omitting degrees higher than 3 and only use from $1 + z + z^2$ terms of the series. Fig. 5 shows to what extend we can rely on the first 3 terms of the series. From this figure it is possible to deduce that if $\frac{2P}{d_1} \le 0.6$, the error remained below 1. It leads to $P < 0.3d_1$. We rewrite (9) as

$$\sum_{i=1}^{n} u_i \frac{1}{1 - \frac{2P}{d_i}} \le 1 \Rightarrow \sum_{i=1}^{n} u_i (1 + \frac{2P}{d_i} + \frac{4P^2}{d_i^2}) \le 1 \tag{11}$$

where $u_i = c_i/d_i$. Thus we have

$$4P^2 \sum_{i=1}^{n} \frac{u_i}{d_i^2} + 2P \sum_{i=1}^{n} \frac{u_i}{d_i} + \sum_{i=1}^{n} u_i - 1 \le 0 \tag{12}$$

Since u_i and d_i are known, it is easy to calculate values of the sums so we are able to simplify (12) as $aP^2 + bP + c \le 0$. Since c is $\sum_{i=1}^{n} u_i - 1$ and it is always negative, $\Delta = b^2 - 4ac$ is positive, hence, resulting values for P are real numbers, and the inequality is solvable. However, one of the roots is always negative and is not acceptable because $0 < P < 0.3d_1$. Hence, the only valid value for P is

$$P \le \frac{-2\sum_{i=1}^{n} \frac{u_i}{d_i} + \sqrt{\Delta}}{8\sum_{i=1}^{n} \frac{u_i}{d_i^2}} \tag{13}$$

To obtain feasible solution, first we calculate P according to (13). If P is larger than $0.5d_1$ and U^R, which is calculated

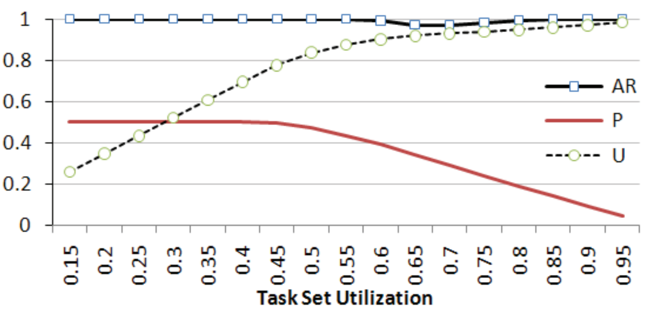

Fig. 6. Admission ratio, normalized period, and resource utilization as a function of task set utilization

from (5), is greater than 1, we try $P = 0.5d_1$. This situation happens in cases where the task set utilization is low and (13) produces relatively large values of P which are not compatible with our other constraints. In this case, we set $P = 0.5d_1$ and check whether U^R is smaller than 1 or not.

If a feasible P is obtained, it is possible to find k_i from (3), o_i from (1), C from (4). Computational complexity of this calculation is $O(n)$ since we have to calculate values of a, b, and c from (12). It is worth mentioning that although P might be close to $0.3d_1$, for other tasks with $d_i \gg d_1$, the ratio between P and d_i is small, hence, $(P/d_i)^3$, $(P/d_i)^4$, etc. become very small values which can be ignored during the calculation of (9). According to Fig. 5, one of the draw backs of our method is that our estimation of $\frac{1}{1-z}$ is not the upper bound. In the future work we try to find an upper bound for this function.

V. Experimental Results

In this section we evaluate the efficiency of FG-PRM regarding admission ratio of the task sets (AR), resource utilization C/P, and normalized resource period, i.e., $\frac{P}{d_1}$. Parameter of the experiments is the utilization of the task set which ranges from 0.15 to 0.95 with step 0.05. For each utilization, we generate 10000 random task sets each with 10 tasks, where their utilization is obtained from uUniFast algorithm. To bound the hyperperiod of the tasks we have considered fixed value $H = 10000$ and we have uniformly chosen value $f_i \in \{1, 2, \ldots, 100\}$ to be able to compute $T_i = H/f_i$, $c_i = u_i T_i$, and $d_i = T_i$.

The average results of AR, P, and U for all utilization values and for all generated task sets have been reported in Fig. 6. AR is obtain by dividing the number of feasible task sets by total number of generated task sets. Horizontal axis of this diagram is task set utilization. As shown by this figure, our method has a significant admission ratio with average 0.99. Also it highly utilizes the resource. In this figure, P is constant before utilization 0.45, and has the maximum value of $P = 0.5d_1$ which leads to normalized P equal to 0.5. From utilization 0.45, P start to decrease which means that condition (13) is activated. The fall of P continues in high utilization task sets. On the other hand, before utilization 0.45 where the period of the resource is constant, by the increase in the utilization of the task set, resource utilization increases linearly.

VI. Conclusion

In this work, a feasible solution for scheduling hard real-time splittable tasks have been presented which is well suited for interactions with non-preemptable devices. The solution is based on the notion of periodic resource model and we have focused on feasible parameter assignment for this model. Using the formulation of reservation task model we have reformulated the schedulability problem. Then the formula is approximated by a solvable polynomial of degree 2. The experiments shown the efficiently of the solution. As a future work, we consider overheads of splitting into account. Also we perform actual experiments on the GPU devices to show the efficiency of our solution on real benchmark applications.

Acknowledgment

The authors would like to thank Mehdi Ghasemi for his helpful comments. This work has been supported by the DAAD scholarship.

References

[1] G. A. Elliott and J. H. Anderson, "Globally scheduled real-time multiprocessor systems with gpus," *Real-Time Systems*, vol. 48, no. 1, pp. 34–74, 2012.

[2] G. Elliott and J. Anderson, "Real-world constraints of gpus in real-time systems," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 2, 2011, pp. 48–54.

[3] D. Grewe and M. OBoyle, "A static task partitioning approach for heterogeneous systems using opencl," in *Compiler Construction*, ser. Lecture Notes in Computer Science, J. Knoop, Ed. Springer Berlin Heidelberg, 2011, vol. 6601, pp. 286–305.

[4] A. Bastoni, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2010, pp. 33–44.

[5] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *IEEE Real-Time Systems Symposium (RTSS)*, 1991, pp. 129–139.

[6] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 287–296.

[7] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *IEEE International Real-Time Systems Symposium (RTSS)*, 2003, pp. 2–12.

[8] N. Fisher and F. Dewan, "A bandwidth allocation scheme for compositional real-time systems with periodic resources," *Real-Time Systems*, vol. 48, no. 3, pp. 223–263, 2012.

[9] N. Fisher and F. Dewan, "Approximate bandwidth allocation for compositional real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2009, pp. 87–96.

[10] T.-H. Lin and W. Tarng, "Scheduling periodic and aperiodic tasks in hard real-time computing systems," *SIGMETRICS Performance Evaluation Review*, vol. 19, no. 1, pp. 31–38, 1991.

[11] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs proportional share resource allocation," in *IEEE International Conference on Multimedia Computing and Systems (ICMCS)*. IEEE Computer Society, 1999, pp. 107–111.

[12] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: Response-time analysis and server design," in *ACM International Conference on Embedded Software (EMSOFT)*. ACM, 2004, pp. 95–103.

[13] J. Theis and G. Fohler, "Transformation of sporadic tasks for off-line scheduling with utilization and response time trade-offs," in *International Conference on Real-Time and Network Systems (RTNS)*, 2011, pp. 119–128.

High-Level Energy Model of Embedded GPU for Real-Time Graphic Rendering

Yu-An Chung, Chen-Wei Huang, and Shiao-Li Tsao
Dept. of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
anthonycj04@gmail.com, cwhuang.cs96g@nctu.edu.tw, sltsao@cs.nctu.edu.tw

***Abstract*—Embedded graphic processing unit (GPU) accelerates a real-time rendering process of a graphics application on mobile devices, however, at the cost of consuming a considerable portion of the system energy [1] which is one of the most critical design issues for battery-operated devices. To estimate the power consumption of a graphics application, conventional approaches collect run-time hardware activities of a GPU, and derive the power consumption of the graphics application based on hardware counters. Unfortunately, these hardware counters and power consumption information are difficult to evaluate from a programmer's point of view. In order to provide graphics programmers a firm notion of how performance and quality relate to energy cost, a high-level power model to assist programmers to balance performance, quality, and energy budget is proposed in this study. Preliminary results demonstrate that the proposed approach is practical and can provide useful information to programmers to optimize the energy efficiency of graphics applications.**

Keywords—Embedded GPU; OpenGL ES, Power Consumption; Power Model

I. INTRODUCTION

Due to high demands on graphics processing, graphic processing unit (GPU) in mobile devices has become an indispensable component. The GPU accelerates the rendering process of a graphics application, however, at the cost of consuming a considerable portion of the system energy [1]. In contrast to desktop developers, mobile device programmers have to strike a good balance between performance, quality, and power consumption in a battery-operated device. It is thus vital for graphics programmers to have a firm notion of how performance and quality relate to energy cost at the development stage.

Desktop GPUs and embedded GPUs are designed differently to suit their working condition and performance target. Desktop GPUs aim for high performance and do not have to worry about power supply. On the other hand, embedded GPUs operate in a battery-operated device and thus have to be designed with low-power consumption. Most of the contemporary embedded GPUs [2, 11, 12] are based on tile-based rendering design (partition the display into small rectangles) to reduce memory transfer energy consumption. Previous studies on GPU power model mainly focuses on desktop GPUs, without considering tile-based design and the associated micro-architecture changes. These design differences have to be considered in constructing an embedded GPU power model. Specifically, we focused on a state-of-the-art Tiled-Based Deferred Rendering (TBDR) architecture proposed by Imagination [2].

Embedded graphics programmers nowadays mainly use OpenGL ES [4] (OpenGL [3] for Embedded Systems) to control the GPU for rendering the scenes in real-time. Therefore we aim to analyze the power consumption behavior of the GPU executing a real-time OpenGL program. An OpenGL program mainly consists of data (such as mesh data, texture data, and camera position) and shader programs (such as vertex shader and fragment shader). Thus our main concept is to develop a micro-benchmark suites specifically tailored for mobile GPUs to systematically stress different pipeline stages (ex. submit lots of vertices to increase vertex shader's loading or even generate pipeline stalls) and analyze the corresponding energy consumption behavior.

According to the analysis results, important parameters are chosen to build our high-level power model. Different from previous studies aiming at low-level hardware dependent desktop GPU power model [5], this research proposes a high-level power model of embedded GPUs from a programmers' perspective. With the proposed power model, graphics programmers will be better equipped with the knowledge about how their programs are being processed by the embedded GPU and the associated energy cost during development time. With such a high-level power model, it is also possible to conduct a high-level resource management to balance between power, performance, and quality for real-time graphic rendering. In summary, the main contributions of this study are:

1. Developed a micro-benchmark suites specifically tailored to the mobile GPU design.
2. Constructed a high-level power model to assist graphics programmer to balance between performance, quality, and energy budget.

The rest of the paper is organized as follows: In Section II, we give a brief summary on previous studies about GPU power models. In Section III, an introduction on the OpenGL pipeline and the architecture of an embedded GPU is given. In Section IV, we discuss the idea of our power model and experiment design. In Section V, the measurement environmental and preliminary experimental results will be shown. Finally, in Section VI, the conclusion and future work will be brought out.

II. RELATED WORK

Previous research on the power consumption of GPU mostly focuses on desktop GPUs with comparatively less focus on embedded GPUs. Collange et al. [6] used an oscilloscope to measure the GPU energy consumption in a

CUDA environment, to find out the bottleneck of a GPGPU program. Shaikh et al. [7] profiled the power consumption of two GPU architectures: GF100 and GT200. Their results show that the power dissipation of a data transfer instruction consumes less than half of that of a kernel instruction. Thus it is possible to identify which part of the program is running at a certain time. Ma et al. [5] chose five main GPU workload signals to build a power model, where the workload signals represent the runtime utilizations of the major pipeline stages on the GPU. They also compared the error rate between two different regression methods, namely Support Vector Regression (SVR) and Simple Linear Regression (SLR). The chosen SVR model outperformed the traditional SLR on their validation datasets. Hong and Kim [8] designed a set of micro-benchmarks to stress different architectural components of the GPU, and built not only the power model of the GPU but also the temperature model as well. They came out with the result: power consumption can be reduced by opening the appropriate number of streaming multiprocessors (SMs) in the GPU instead of using all the SMs. Leng et al. [9] built a power model for GPGPU using the power measurement data and performance counters from GPGPU-Sim, and also can estimate the GPU component's power consumption. They proposed a micro-benchmarking design methodology which includes the following: component stress, access patterns and test coverage. The above studies are all based on GPUs on desktop computers. Also, most of the above studies require hardware performance counters to estimate the power, which might not be easy to be interpreted by graphics programmers.

Following we list some studies related to mobile GPUs. Mochocki et al. [10] used three embedded processors to simulate different stages in the 3D pipeline. They analyzed how the factors (resolution, frame rate, level of detail, lighting model, and texture model) affect the 3D pipelines to result workload variations and imbalances. Moreover, DVFS was applied to processors to reduce the workload imbalance and can achieve up to 50% energy saving. Vatjus-Anttila et al. [1] built a power model based on three render complexity characteristics: number of triangles, render batches and addressed texels. Instead of measuring only the GPU's power consumption, the whole device's power consumption was used. To compensate the overestimated power, they empirically deducted 45% of the consumption based on the ad-hoc hypothesis that 50% of the 3D content could be left unacknowledged due to the back-face triangle culling, and 10% due to depth testing. Mochocki et al. [10] studied about how some graphics factors affect the 3D pipelines, but they did not use a real embedded GPU for their experiments and neglected the architectural differences between desktop and embedded GPUs. Vatjus-Anttila et al. [1] built a power model for the whole embedded system based on render complexity. Our goal is to first understand the relation between high-level graphics parameters and the graphics pipelines. Then, we build a high-level power model for embedded GPU that only requires high-level parameters to estimate the power for real-time graphic rendering.

III. Background

A. OpenGL Pipelines

OpenGL is an API for advanced 3D graphics and it provides functionality to control the GPU, while OpenGL ES is specially targeted at handheld and embedded devices. The OpenGL graphics pipelines are shown in Fig. 1. Mesh data are first sent to the GPU, then vertex shading and primitive assembly are done on each vertex. Primitives are converted into fragments in the rasterization phase, and the fragment shader then either discards a non-visible fragment or generates a color for a visible fragment. The per-fragment stage goes through a series of tests (scissor test, stencil and depth test, and blending) and writes the resulting color into the frame buffer. In an OpenGL program, we provide some input, including mesh data, vertex and fragment shader program, texture data, etc. for the GPU. The GPU then takes all the information about the 3D objects and renders it on screen. In this paper, we alter those high-level inputs and see how the GPU hardware reacts.

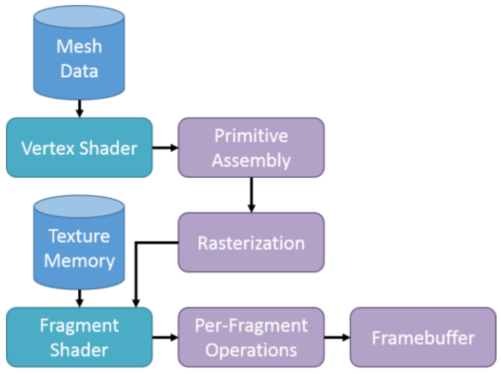

Fig. 1. OpenGL pipeline.

B. Embedded GPU

Desktop GPUs and embedded GPUs are designed differently to suit their working condition and performance target. Desktop GPUs aim for high-performance whereas embedded GPUs target for low power.

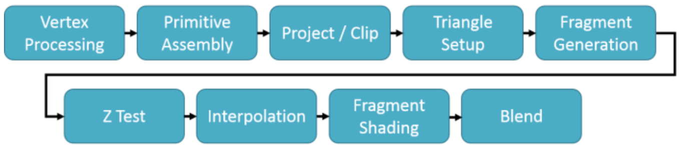

Fig. 2. Immediate Mode Rendering (IMR).

The desktop GPU goes through the Immediate Mode Rendering (IMR) pipeline as shown in Fig. 2. Under IMR, each submitted object goes through the entire pipeline independently until the very last stages. Hence IMR enables processing with the maximum parallelism and speed. However, there are two weakness in this design, namely overdraw (fragments not shown in the final display are still processed) and large unnecessary memory transfers for fetching data associated to these unused fragments. This will bring heavy burdens for the battery-operated mobile devices.

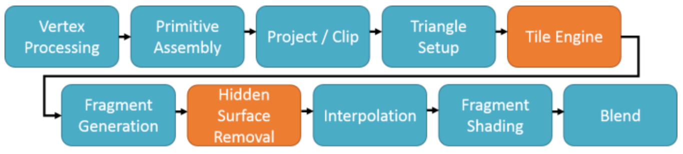

Fig. 3. Tile-based deferred rendering.

In order to solve these two critical issues, most of the contemporary embedded GPUs [2, 11, 12] adopt the tiling-based architecture. Since the data transfer between system memory and the GPU is one of the biggest cause for the power consumption of the GPU, tiling design claims to be able to reduce memory bandwidth requirement by partitioning the frame into small rectangles of pixels before rasterization. After coordinate transformation and triangle setup, the GPU determines tile-coverage of each triangle and records this information in a per-tile list. With this per-tile information, only relevant geometry data is needed when processing each tile in subsequent stages, therefore lowers the system memory bandwidth significantly. Imagination PowerVR further adopts the Tile Based Deferred Rendering (TBDR) [2] pipeline as shown in Fig. 3 to further reduce both overdraw and unnecessary memory transfers. We aim to verify the effectiveness of reducing overdraw and memory transfer of both TBDR and tiling-based designs in our study.

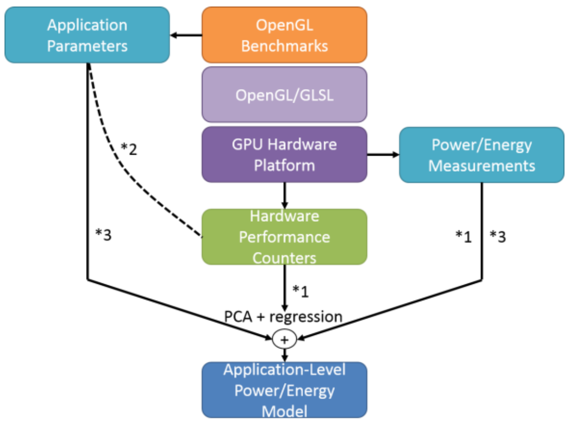

Fig. 4. Construction of high-level power model.

IV. Methodology

Previous studies on GPU power models [1] mostly focus on relating the power consumption to the hardware events by observing hardware performance counters of the GPU. A set of important hardware events such as memory transfer, cache misses, shader utilization, texture access, etc. were chosen and they are trained by a benchmark suite to build up the power model (*1 in Fig. 4). This kind of power models are not very intuitive to graphics programmers since the programmers usually do not have a direct feeling about how their programs translate into hardware events. Our goal is to build a power model that can estimate power consumption with graphics related high-level parameters that programmers are familiar with (*2 in Fig. 4). Using principal component analysis (PCA) [13] to select high-level parameters that are vital to the energy consumption, we can statistically relate the high-level parameters to GPU power consumption (*3 in Fig. 4).

With this power model, programmers can have a better notion on how the GPU responds to their graphics program and can achieve a balance between performance, quality and energy.

TABLE I. Hardware performance counter.

Hardware Counter	Description
USSE load: vertex	Percentage of time that the Universal Scalable Shader Engine (USSE) has spent processing vertices.
TA load	Percentage of time that the Tile Accelerator (TA) unit is busy. The TA unit is responsible for clipping, projecting, culling and tiling transformed polygons.
ISP load	The load of the Image Synthesis Processor (ISP) unit. The ISP is responsible for executing the per-tile Hidden Surface Removal. It also performs the depth and stencil operations for the tile using the GPU's on-chip memory.
TSP load	The percentage of time that the Texture and Shading Processor (TSP) unit is busy. The job of TSP is to schedule fragment processing tasks, iterations, and texture data pre-fetch.
USSE load: pixel	Percentage of time that the Universal Scalable Shader Engine has spent processing pixels.

As mentioned in Section III, an OpenGL program receives a set of input data, including mesh data, texture data, vertex and fragment shader programs and some other control data. First, we design micro-benchmarks to find out how high-level parameters affect the GPU components (*2 of Fig. 4). The experiment environment is PandaBoard using the OMAP4430 processor with Imagination PowerVR SGX540 GPU [14]. Test programs are executed under Ubuntu 11.10 with a 3.1.0 Linux kernel using the OpenGL ES 2.0 library. The description of the hardware performance counters we chose are listed in Table I. These hardware counters are chosen because they represent the runtime utilizations of the major pipeline stages of the GPU. After the relationship between high-level parameters and GPU components are known and understood, we can then build the power model according to the experimental results (*3 of Fig. 4).

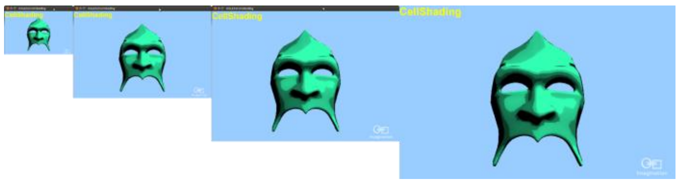

Fig. 5. Test program for resolution.

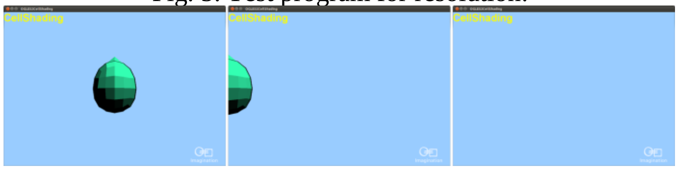

Fig. 6. Test Program for position of object.

Currently, we conduct two experiments to analyze how high-level input affects the GPU's behavior. In each experiment, we only alter one input parameter while keeping the others fixed. The first test program alters the resolution of the program, while leaving the mesh data, texture data, vertex and fragment shader unchanged. As shown in Fig. 5, four resolutions (360x225, 720x450, 980x675, and 1440x900) were tested. Since the number of tiles processed by the GPU increase linearly with the resolution, we aim to see the effect of number of tiles to the GPU's behavior. The second test program changes only the position of the object. As shown in Fig 6, the object was placed in the center of the screen showing, 100% of the object; placed at the edge of the screen, showing around 50% of the object; and placed outside of the screen, showing 0% of the object. This test will affect the number of vertices left after clipping, since the vertices that are out of viewing frustum will be clipped out. With the above two experiments, we are able to observe how the GPU reacts to the change on resolutions and vertices after clipping and culling. Similarly, we will further explore other relevant events that are important to the graphics programmers in the future.

V. Preliminary Results

A. Resolution

In the first experiment, the same graphics program is executed with various resolutions. Table II shows the time spent on each GPU components. In Table II, we can see that the time spent on vertex shading under different resolutions are nearly constant. This is because the same object is submitted to the vertex shader, and therefore the same amount of calculation is needed for the vertex shader. When the resolution increases, more tiles are needed to be processed. Therefore the TA will have to spend more time on the tiling operation. Similarly, the rest of the pipeline after TA has more workload due to the increased number of tiles.

TABLE II. Experimental results for resolution tests.

Resolution	**Execution time**				
	USSE: vertex time (ms)	**TA time (ms)**	**ISP time (ms)**	**TSP time (ms)**	**USSE pixel time (ms)**
360x225	30.53975	119.1380	67.48887	172.7195	231.2407
720x450	25.58628	184.9845	116.0092	499.6496	731.0922
980x675	26.30886	298.9563	141.1661	954.9022	1391.347
1440x900	24.77687	476.6062	242.2432	1691.457	2469.837

TABLE III. Experimental results under different positions of objects.

# of vertices after clipping and culling	**Execution time**				
	USSE: vertex time (ms)	**TA time (ms)**	**ISP time (ms)**	**TSP time (ms)**	**USSE pixel time (ms)**
7869	367.231	11735.85	8222.787	1356.312	1915.855
3805	340.866	6372.385	3855.828	1108.976	1553.598
51	365.498	1089.121	264.1725	1199.218	1690.872

B. Position of Object

In the second experiment, we keep the mesh data, shading programs, texture and resolution intact with focus on changing only the position of the object. Table III shows the same information as Table II. Since the same object is submitted, we can see that the time spent on vertex shading remains relatively stable in Table III. When moving the object to the border of the screen or even out of the screen, vertices will be clipped by the TA. Clipping effectively reduce the amount of visible fragments which enables the TA to process faster in handling each tile. Similar benefit is enjoyed by the ISP stage. Since we keep the same resolution in this experiment, the workload of TSP and USSE spent on processing the pixels is also relatively stable.

VI. Future Work

In the future, we will design a complete methodology consisting of a micro-benchmark suite and statistic method to analyze the importance of the high-level graphics parameters on the embedded GPU activities. This will help graphics programmers to understand how the graphics programs relate to the GPU power consumption. With the power model, we can conduct high-level power management which can adaptively achieve a balance between rendering performance, quality, and energy for real-time graphic rendering.

VII. References

[1] J. M. Vatjus-Anttila, T. Koskela, and S. Hickey, "Power Consumption Model of a Mobile GPU Based on Rendering Complexity," in Proceedings of the 2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies, Washington, DC, USA, 2013, pp. 210–215.

[2] ImaginationTechnologies Ltd, "PowerVR Series 5 Architecture Guide for Developers." 2014.

[3] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Addison-Wesley Professional, 2013.

[4] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL(R) ES 2.0 Programming Guide*, 1st ed. Addison-Wesley Professional, 2008.

[5] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical Power Consumption Analysis and Modeling for GPU-based Computing," in *Proc. of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[6] S. Collange, D. Defour, and A. Tisserand, "Power Consumption of GPUs from a Software Perspective," in *Proceedings of the 9th International Conference on Computational Science: Part I*, Berlin, Heidelberg, 2009, pp. 914–923.

[7] M. Z. Shaikh, M. Gregoire, W. Li, M. Wroblewski, and S. Simon, "In Situ Power Analysis of General Purpose Graphical Processing Units," in *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Washington, DC, USA, 2011, pp. 40–44.

[8] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010, pp. 280–289.

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2013, pp. 487–498.

[10] B. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D Graphics," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, 3001 Leuven, Belgium, Belgium, 2006, pp. 502–507.

[11] ARM, "ARM Mali GPU - OpenGL ES Application Optimization Guide." 2013.

[12] Rob Clark, "Adreno tiling." Internet: https://github.com/freedreno/freedreno/wiki/Adreno-tiling, Apr-2014.

[13] I. T. Jolliffe, *Principal Component Analysis*. New York: Springer Verlag, 2002.

[14] PandaBoard: http://pandaboard.org/

Integration Framework for Legacy and Generated Code in MBD

Atsushi Ohno*, Takayuki Hikawa†, Nobuhiko Nishio†, Takuya Azumi‡
*Graduate school of Information Science and Engineering, Ritsumeikan University
†College of Information Science and Engineering, Ritsumeikan University
‡Graduate School of Engineering Science, Osaka University

***Abstract*—On-board software becomes large-scale and more complicated, as an electronic control system and high functionalities of automobile progresses. Model Based Development (MBD) has appeared as a technology that control designers create and verify control models on MATLAB/Simulink. The control designers, however, do not consider the software development in the control design phase. It causes differences of interfaces between driver code and automatically generated code. In particular, some required information for adjusting I/O relations does not exist on the code description. Thus, a software development workload has been increased and human errors may increase. This paper presents an integration framework for treating automatically generated code as a component on a component based system and producing a wrapper component based on interface information from a command line. Integrated code running on an actual machine is demonstrated and it contributes to improve the automation efficiency.**

***Keywords*-Automobile, Component-Based Development, Model-Based Development, Legacy code, Code integration**

I. Introduction

Recently, as the electronic control system of automobile progresses, the number of Electronic Control Unit (ECU) required in automobile and code lines of on-board software are increasing rapidly. Embedded systems have become a large-scale and more complicated. Thus, it is difficult for software developer to create the software that meets control designer requirements.

Model-Based development (MBD) has paid attention as a method to make control design process more efficient. In particular, it becomes to be implemented in automobile developments. Control designers design controller models with CAE tools such as MATLAB/Simulink. Hence, they can share the behavior of the model each other.

The MBD tools generate code based on control design models and ensure that the generated code is the same as what the controller designer intended. However, it does not mean that all the code used on the software is generated by MBD tools. Whole code consists of OS, middleware, I/O driver and generated code. In terms of OS, middleware, and I/O drivers, legacy code has been used. Therefore, software developers must integrate the legacy and generated code. Developers are required to match the interface of the legacy code and the automatically generated code manually. There are variable types, increments treated per bit, and mapping. In particular, required information for converting the increments and mapping do not exist on the code description. Thus, a complete automation is difficult. In a large-scale and complicated embedded system development, a workload of software developers has been increased and thus human errors may increase.

The purpose of this paper is to reduce workloads of software developers by proposing a framework for semi-automatically integrating generated code with legacy code. On an integration process of a conventional development, it is necessary to fill the difference of interface between generated code and legacy code. Therefore, the software developers make wrapper code and integrate them manually to fill the difference. After the implementation of MBD, the wrapper code is still necessary. This is because an increment treated per bit of input and output are sometimes different between the automatically generated code control designers create using the tool and existing driver code software developers handle. The framework has a mechanism which automatically generates wrapper code converting increment treated per bit only by interactive exchange on a command line and minimizes the human intervention during integration using component based system. Component-Based Development (CBD) is a technique for building software with composing as parts divided by function. The CBD contributes to increasing visibility and reusability. Component based framework for embedded systems have been proposed such as TECS [1], THINK [3], SaveCCM [2], and SmartC [4]. The reason for using the component based framework is easy to manage each interface as an integration support.
The contributions are as follows:

- reduction in the workloads of software developers
- reduction in human errors occurred during the integration process
- solving the differences of interfaces between the control design and software development phases

The rest of this paper is organized as follows. This paper discusses conventional MBD in Section II, and Section III describes the design goals. Section IV presents design and implementation. Section V evaluates the framework, and then Section VI concludes.

II. Conventional MBD

An example of an MBD control design flow of inverted pendulum is shown below. Controlled object called “puppy” is a parallel two-wheel type robot. First, control designers make a controlled object model itself, such as differential equations expressed in physical equations. In addition, the control designers create a controller model of an inverted pendulum using a control theory. A controller determines an output for the controlled object based on its

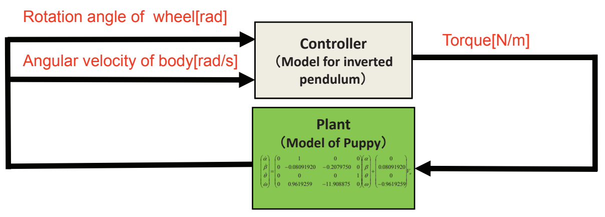
Rotation angle of wheel[rad]
Angular velocity of body[rad/s]
Controller (Model for inverted pendulum)
Torque[N/m]
Plant (Model of Puppy)

Figure 1. Control design model

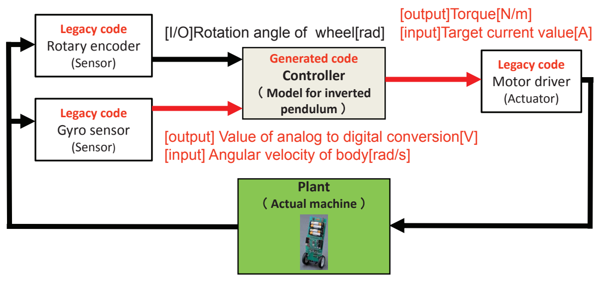
Legacy code
Rotary encoder
(Sensor)
[I/O]Rotation angle of wheel[rad]
[output]Torque[N/m]
[input]Target current value[A]
Generated code
Controller
(Model for inverted pendulum)
Legacy code
Motor driver
(Actuator)
Legacy code
Gyro sensor
(Sensor)
[output] Value of analog to digital conversion[V]
[input] Angular velocity of body[rad/s]
Plant
(Actual machine)

Figure 2. I/O relation

state. Using these numerical formulas and controller models, the control designers perform simulation on CAE tools such as MATLAB/Simulink. Figure 1 shows an example of a control design model created by control designers in MBD. The controller outputs a torque value of motor based on a motor turning angle and an angular velocity inputs from the controlled object. Simulink Coder is a code generator of MATLAB/Simulink extension. It generates C code based on the controller model. The generated code enables software developers to use a control algorithm working as designed.

The control design model shown in Figure 1, however, does not consider the implementation environment. Figure 2 describes a re-designed model of Figure 1 and taking the implementation environment into consideration. A numerical formula model is replaced with an actual machine. In addition, three models are added. A first model is the gyro sensor model which outputs an inclination of a body as a gyro sensor value. A second model is the rotary encoder model which outputs rotational angles of the wheels. The last model is the motor driver model which controls a motor using an input of a target current value for the motor. Legacy code previously developed or provided by vendors is applied for an implementation program of the gyro sensor, rotary encoder, and motor driver models.

I/O relations in Figure 2 indicate that output of the rotary encoder and input of the controller have the same unit of rotational angles of the wheels. The input of the controller, however, is the body angular velocity and output of gyro sensor is the AD conversion value of it. Therefore, the unit does not match. Non-consideration of coding during control design phase cause this problem.

In order to match input with output in the I/O relations, additional manually-created code is required. Hence, it enables the software developers to use generated code as shown in Figure 3. In the control design phase, an ignorance of implementation resulted in making the model shown at Figure 1. As a result, the software developers must change interface of legacy code in a software devel-

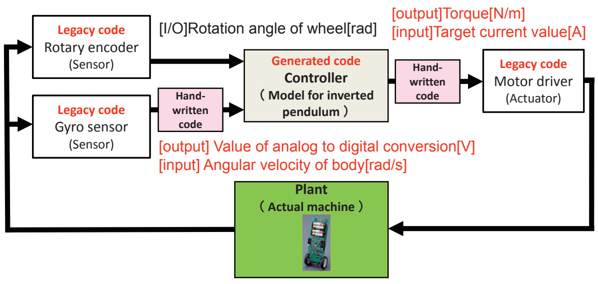

Figure 3. Implementation model

opment process shown in Figure 3. Converting the smallest unit that can be represented per bit and mapping a input value to other data are given as an operation for adjusting I/O relations. Details of each element are described in below. On an operation of a Least Significant Bit (LSB) conversion, minimum unit of input increment information per bit is adjusted to the unit of output. The control system may use a value by converting the voltage such a torque or an angle value to other value for being treated as an argument. While converting, a minimum unit treated per bit on arguments is called "LSB". LSB is defined as below:

$$LSB = \frac{maximum\ voltage - minimum\ voltage}{maximum\ argument\ value - minimum\ argument\ value} \quad (1)$$

On an operation of mapping a data to other data, the software developers write the code which converts input value to other values as an output. For instance, as shown in Figure 3, the gyro sensor outputs an AD conversion value while an input for the controller assumes the angular velocity of the vehicle. In this conversion, hand-written code is required to provide functions which outputs 0 when an AD conversion value of a gyro sensor is an infinitesimal value, and otherwise outputs a converting value multiplied the AD conversion value and a conversion factor. Thus, the software developers must write code when the I/O relations of control design phase is different with that of implementation phase. In particular, control designers set maximum and minimum voltage of LSB on their own terms. The software developers need to communicate with them or check specifications in order to convert LSB. It indicates that using manpower is unavoidable while converting LSB. Therefore, it is a challenge to make code integration automated while keeping the use of manpower at the minimum.

III. Design Goals

The following problems are required to solve in regard to the integration process of conventional MBD:

A. *Integration with legacy and generated code*

The framework enables the software developers to integrate and generate code produced from a control design tool, and legacy code and utilize them. It also enables them to generate wrappers automatically as much as possible.

B. *Providing a wrapper of LSB conversion*

The framework provides a function to convert the LSB difference between the input and output increment per bit when software developers integrate the generated code with legacy code. The wrapper converts the LSB value of calling function to the LSB value of providing

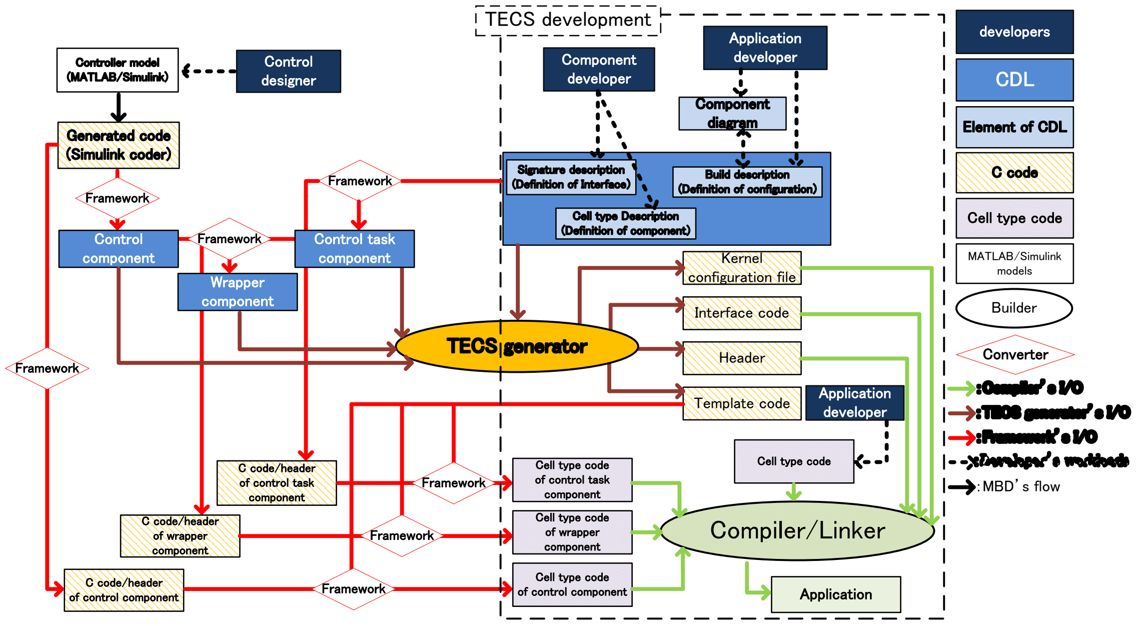
TECS development
Controller model (MATLAB/Simulink)
Control designer
Generated code (Simulink coder)
Framework
Control component
Wrapper component
Control task component
Component developer
Application developer
Component diagram
Signature description (Definition of interface)
Build description (Definition of configuration)
Cell type Description (Definition of component)
TECS generator
Kernel configuration file
Interface code
Header
Template code
Application developer
Cell type code
C code/header of control task component
C code/header of wrapper component
C code/header of control component
Cell type code of control task component
Cell type code of wrapper component
Cell type code of control component
Compiler/Linker
Application
developers
CDL
Element of CDL
C code
Cell type code
MATLAB/Simulink models
Builder
Converter
:Compiler's I/O
:TECS generator's I/O
:Framework's I/O
:Developer's workloads
:MBD's flow

Figure 4. Integration flow

function. In order to warn the software developers that there is a conversion error after the conversion process, the framework needs a warning system.

C. Providing a wrapper of mapping function

The framework also provides a function to convert a value of calling function to a value of providing function by the conversion method an user selected.

IV. Design and Implementation

The framework integrates legacy components with a component utilizing automatically generated code. The Code generation tool is Simulink Coder. TECS is used as a component based framework on this study and overview of it is examined below.

A. Overview of TECS

Development flow of TECS is shown in dashed line enclosing the image in Figure 4. TECS which features static binding and C based development is suitable for embedded systems [1]. The reason of using TECS on this framework is that TECS generator is available. The TECS generator generates code automatically based on Component Description Language (CDL). The code includes kernel configuration files, interface code, header, and template code. It contributes to reduce a labor of managing interfaces, such as data type during compile-time and link-time.

B. Integrative approach

A component has entry port and call port interfaces based on the TECS component model [1]. The entry port is an interface to provide services to other components. The service of the entry port is called the entry function. The call port is an interface to use the services of other components. Based on the above, legacy and generated code integration flow is examined and shown in Figure 4. First, the framework makes the generated code available as a component. This component is defined as "control component". Control component's interfaces are input and output of control design model. Second, the framework makes control component available as a cyclic task. In addition, it makes a function of control component executable from legacy code. The component is defined as a "control task component". An argument value as an interface of control component is required for generation of the control task component. Finally, software developers confirm the differences of the interfaces between the control component and the control task component. If the interfaces are different, software developers generate a wrapper component, and after that these components are integrated on TECS. Wrapper components provide functions converting an input value to other value as an user selected form. LSB and mapping components are described below.

1) LSB component: This component provides function converting input resolution to output resolution. As mentioned above, LSB is defined in equation 1. Difference of LSB between input and output is a problem software developers cannot find from the implementation model. Therefore, software developers enter LSB control designers sets and LSB software developers sets from terminal. For LSB conversion, the framework generates code of LSB wrapper component. Converting LSB is calculated based on equation 2. 'y' means voltage value which software developers or control designers consider, and 'x' means an argument value. In addition, minimum value between a voltage value and argument value may be different. Therefore, an intercept which compensates the difference is defined 'I'.

$$y = x \times LSB + I \tag{2}$$

On software, argument value 'x' is needed. 'y_1' is a voltage value of a call port and 'y_2' is a voltage value of an entry port. $y_1 = y_2$ is expressed in equation 3.

Table I
COMPARING THE RESULT

	A		B		C		**D**		E		F		G	
	Former	After	Former	After	Former	After	Former	After	Former	After	Former	After	former	After
CDL	8	0	28+2α	0	6+3α	0	15	0	11	0	18+3γ	0	12	0
C code & header	10+α+β	0	1+α	0	1+α	0	1	0	5	0	5+2γ	0	1	0
Total	18+α+β	0	29+3α	0	7+4α	0	16	0	16	0	23+5γ	0	13	0

Table II
CORRESPONDING TABLE

Letter	Required description for each component
A	Control component
B	Control task component
C	Modification of control component for wrapper component
D	LSB conversion component
E	Mapping component using all data
F	Mapping component using inflection point
G	Mapping component using proportional relations

$$x_1 \times LSB_1 + I_1 = x_2 \times LSB_2 + I_2 \qquad (3)$$

This framework is designed and implemented to convert the call port value 'x_1' adjusting the entry port. 'x_2' is calculated by equation 4.

$$x_2 = \frac{x_1 \times LSB_1 + I_1 - I_2}{LSB_2} \qquad (4)$$

2) Mapping component: This component provides functions converting an input value to other value, and output the other value by corresponding relationship of data prepared in advance. There are situations when complex calculations are needed to convert data value. In the case, data conversion requires significant processing. It could influence optimization of control. Therefore, the framework provides three kinds of components as below:

A method using all data: This method is preparing a corresponding relation of all data between the call port and the entry port in advance. Hence, calculating processing can be skipped and it reduces an executing time.

A method using inflection point: In this method, correspond value is calculated based on input value using linear functions. Linear functions exists between inflections points. Therefore, processing time increases although memory area is decreased, compared to the method using all data. Based on information of inflection points the user selects, the linear function is determined and it calculates a value of the entry port. As for this method, equivalent value may not represent. In this case, the framework must warn the user to announce the extent of an error.

A method using proportional relationship: This component uses a coefficient of proportion for mapping output value. In case of proportion such as relationships of electric current and voltage, electric current, and torque, this mapping component is efficient.

V. EVALUATION

Evaluation environment is examined in Section II and IV. A demonstration shows that an actual machine runs inverted pendulum after the code integration on the framework. The framework is evaluated in terms of automation and efficiency. Greek characters in Table I means as following. 'α' expresses the number of I/O in the Model. 'β' describes lines of code copied from automatically generated code. 'γ' presents the number of inflection points. As shown in Table I and Table II, an amount of hand-coding is reduced while integrating. Software developers, however, input LSB and mapping information into terminal for generating each component. Software developers do not have to consider how to implement software, and only need to consider the differences of interfaces compared to coding. Therefore, it indicates that the workloads of the software developers can be reduced by the proposed automation. Next, hand coding may cause human errors and an amount of hand-written code decreasing indicates its reduction. Third, LSB and mapping components runs on an actual machine without any problem. It shows handling the difference of I/O relations between the control and software design phases.

VI. CONCLUSION

This paper described the framework aimed at efficient integration for legacy and generated code in MBD. Verification of an actual machine and evaluation of automation have been done, and they show the usefulness. As for reduction of software developer workloads, the framework contributes providing a mechanism of integration automatically. Regarding human errors reduction, it contributes providing wrapper components. In terms of handling the difference of the interfaces, it also contributes providing LSB and mapping components. The future work is supporting OS functionalities assigning an appropriate priority and a periodic task on the framework.

ACKNOWLEDGMENT

This research was partially supported by JSPS KAKENHI (#40582036).

REFERENCES

[1] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada. A new specification of software components for embedded systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 46–50, May 2007.

[2] J. Carlson, J. Hakansson, and P. Pettersson. Saveccm: An analysable component model for real-time systems. *Electronic Notes in Theoretical Computer Science*, 160:127–140, 2006.

[3] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *2002 USENIX Annual Technical Conference*, pages 73–86, 2002.

[4] G. Yang, H. Li, and Z. Wu. Smartc: A component-based hierarchical modeling language for automotive electronics. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 203–210. IEEE, 2006.

Overrun-freeness verification of Rate-Monotonic Least-Splitting Real-Time Scheduler on Multicores

Mahmoud Naghibzadeh and Amin Rezaeian
Department of Computer Engineering
Ferdowsi University of Mashhad, Mashhad, Iran
naghibzadeh@um.ac.ir, amin.rezaeian@stu-mail.um.ac.ir

***Abstract*—In real-time task scheduling, semi-partitioning allows some tasks to be split into portions and each portion to be assigned to a different core. This improves the performance of system but by counting each portion as a separate task it increases effective number of tasks to be scheduled. This research suggests a semi-partitioning method and assigns each partition to a separate core to be scheduled by the well-known scheduler called Rate-Monotonic (RM). To assure non-concurrent execution of portions of a task, there is no need to define release time for any portion. It is theoretically proven that with the proposed semi-partitioning and RM scheduling, all cores always run their tasks overrun-free. Besides, experimental results show that overall system utilization is noticeably boosted and also number of broken tasks is not higher than the best RM-based methods.**

keywords: rate-monotonic least splitting, semi-partitioning, hard real-time scheduling

I. INTRODUCTION

A multicore system is composed of several processing elements, called cores, in which all cores can do their processing in parallel. They all share the same main memory but each can have its own private cache memory. With this structure, a sequential computation can be shared among many cores if not more than one core is executing the computation simultaneously [1]. While manufacturers tend to use multicore processors in new artifacts, software facilities to use all available power of multicores are yet to develop [2]. Scheduling algorithms play a significant role in overrun-freeness verification of hard real-time systems, i.e., making sure that every request is executed before its deadline. However, being multiprocessor/multicore adds a new dimension to the analysis; how to assign tasks or their requests to different processors/cores.

In this paper, the problem of scheduling periodic hard real-time task sets with implicit deadlines, i.e., when the relative deadline of a request is equal to its minimum request interval, on multicores is investigated. One way of categorizing scheduling methods for multicores is global, partitioned, and semi-partitioned, categories. In global scheduling, there is only one queue (or pool) of requests and each core takes its next request for execution from this queue. In partitioned, the set of tasks are divided and each partition is assigned to a separate core. Finally, in semi-partitioned, some tasks are wholly assigned to specific cores and some tasks are shared among more than one cores, with the restriction that not more than one core can work on a request of the shared task, simultaneously.

It is usually the case that semi-partitioned scheduling leads to a higher overall utilization of the whole system than global scheduling, for both fixed-priority and dynamic priority. However, partitioning is a time consuming task which is computationally equivalent to bin-packing problem that is known to be an NP-hard problem [3]. The good side of it is that portioning is done off-line. Therefore, for small number of tasks the time taken by partitioning is tolerable, but for large number of tasks efficient heuristics are thought. A semi-partitioned approach binds a disjoint set of whole tasks to each core and lets remaining tasks be executed on multiple cores while everyone's share is defined. In one of the researches on semi-partitioned methods in which Rate-Monotonic (RM) scheduler is used in each processor, worst case utilization is reported to be 0.693 [4].

In this paper, a different semi-partitioned scheduling algorithm called Rate-Monotonic Least Splitting (RMLS) is proposed for multicores. The scheduler of each core is basically RM with very minor changes to avoid simultaneous execution of a shared task by more than one processor. Using this algorithm, the number of split tasks is at the most equal to number of used cores minus one. Besides, no task is split in more than two portions. Splitting fewer tasks has two benefits, (1) effective number of tasks in the Liu and Layland's bound, i.e., $\Theta(n)=2(2^{1/n}-1)$, is reduced which in turn (2) increases overall system utilization.

The following notations are used throughout the paper. n: total number of tasks, n_1: total number of task and subtasks, m: total number of available cores (or processors), m_1: total number of used cores, τ_i: ith task, T_i: minimum interarrival time between any two consecutive requests of task τ_i, C_i: maximum computation time needed by every request of task τ_i with $C_i \leq T_i$, and finally u_i: the utilization of task τ_i which is equal to Ci/Ti.

In Section 2 related work is briefly reviewed: Section 3 describes the proposed RMLS semi-partitioned scheduling, Section 4 is the theoretical foundations and overrun-freeness proof of the algorithm, in Section 5 the algorithm is simulated and results are documented, and finally a summary and future work is presented in Section 5.

II. RELATED WORK

Many researchers have studied the semi-partitioning problem with Earliest Deadline First (EDF) scheduling [5-7].

The best known worst-case utilization bound using semi-partitioned EDF scheduling on multicores is 65% for Earliest Deadline Deferrable Portion (EDDP) algorithm [8]. Later, they proposed EDF with Window-constraint Migration (EDF-WM) which has less context switch overhead [9]. The NPS-F is a configurable method that has a tradeoff parameter between utilisation and preemptions [10]. On the other hand, relatively fewer algorithms are proposed for fixed-priority algorithms [11]. Rate Monotonic Deferrable Portion (RMDP) and Deadline Monotonic with Priority Migration (DM-PM) fixed-priority algorithms are proposed by Kato et al [12, 13]. The worst-case utilization bound of those algorithms is 50%. The concept of *portion* and how a shared request migrates between two cores is explained in the same references. PDMS_HPTS_DS is proposed by Lakshmanan et al. [2] which reaches 65% utilization. This bound can be extended to 69.3% for *light* tasks, i.e., tasks with utilizations less than 0.41. Guan et al. proposed two algorithms called SPA1 and SPA2 [4, 11]. SPA2 has a pre-assignment phase in which special *heavy* tasks are assigned to processors, first. The number of split tasks is m-1 and SPA2 reaches the worst-case utilization bound of 0.693. This is equal to the Liu and Layland bound [14] for single processor systems. However, the worst-case bound in SPA2 is calculated using n which is the cardinality of the whole task-set, and every processor's utilization must be less than or equal to that. For further reading on real-time scheduling algorithms and related issues refer to [15].

III. Semi-partitioned RMLS

Basic idea of the semi-partitioned method which is being presented here is presented in workshop [16]. There, the fundamental theorem which guarantees the overrun-freeness of system was not proven. In addition, none of the other theoretical results provided by this paper have appeared in that paper. A brief introduction of the method is repeated here and new findings and performance evaluations follow. The method is called Rate-Monotonic Least splitting (RMLS) because it is a semi-partitioned method in which only m_l-1 tasks are split.

Our experiments show that achieved processor utilization is higher than the best known results for general real-time systems, i.e., no restrictions on utilization of individual tasks, running with fixed-priority schedulers up to now. The proposed assignment algorithm is composed of two steps, see Algorithm 1.

In Step 1 (Lines 1 to 9), all pairs of tasks, τ_i, and τ_j, with total utilizations satisfying $\Theta(3) \le U_i + U_j \le 1$ are found and each pair is assigned to a separate processor. Meanwhile, heavy tasks, i.e. a task τ_l with $U_l \ge \Theta(2)$, are recognized and each such task is assigned to a separate processor. The scheduler of each core with two tasks is taken to be Delayed Rate Monotonic (DRM) which is a modified version of RM. Details of how DRM works are explained in [17]. Any system composed of two tasks with utilization less than or equal to one can run overrun-free with DRM. The scheduler of all other sets will be the conventional RM.

Step 1 serves two purposes: (1) it increases the number of cores with high, and (2) it increases the number of processors with no split task, i.e., decreases the total number of split-tasks.

```
Data: Task-Set
Result: Processor assignments
1  Find tasks with largest and smallest utilizations;
2  While smallest and the largest tasks are different
3     If it's worth assigning them to a separate processor
4        do so and find the next largest and smallest;
5     Else if it's worth to assign largest task to a separate
6        processor do so and find the next largest;
7        Else if sum of both utilization is too large
8              discard current largest and find next largest;
9              Else discard current smallest and find next smallest;
11 Take an unassigned processor as current-processor
12 While there is an unscheduled task
13    Find the unscheduled task with highest priority as
          current-task and assign it to current-processor;
14    If current-processor is not overrun-free
15       Remove the task with least loss from current-processor as
            split-task, split it and assign its first-portion to
            current-processor;
16       Take a new processor and make it current-processor and
            assign the second-portion of the task to it;
```

Algorithm 1. Packing algorithm

In step 2 (Lines 11 to 16.), all unassigned task are sorted in decreasing order of RM priorities, i.e., non-descending order of their request interval lengths. An empty core is picked and starting from the first unassigned task, tasks are assigned to the core one at a time until the current task, say task τ_i, will make the core overloaded. Task τ_i is also assigned to the core but one of the assigned tasks, except the one which is shared with the previous core, is selected to be split and shared with the next core. The split task may happen to be τ_i. In the following example a scenario is explained and it is clarified what criteria is used to select a task to be split. The selected task is split into two subtasks such that the first subtask is assigned to the current core and makes it full with respect to Liu and Layland's bound for the respective number of tasks and subtasks in this processor.

A new core is taken and the second portion of the current split task is assigned to it. The process of assigning tasks to cores continues until all tasks are assigned. If there are enough cores the assignment successfully complete.

Example 1: suppose the current core is p_k and task τ_i is the task which is split into two portions τ_{i1} and τ_{i2} with execution times C_{i1} and C_{i2}, respectively. The utilization of τ_{i1} is $u_{i1} = \frac{C_{i1}}{T_i}$ for core p_k. A new core, p_{k+1}, is taken and the second portion of task τ_i, τ_{i2}, is assigned to this core. Although the *actual utilization* of this portion is $\frac{C_{i2}}{T_i}$, its *effective utilization* on core p_{k+1} is taken to be

$$u_{i2} = \frac{C_{i2}}{T_i - C_{i1}} \quad (1)$$

This is because, in the worst case, a request from subtask τ_{i2} will have only T_i-C_{i1} time to be executed. Effective utilization of the subtask is always greater than or equal to its actual utilization. Therefore, $UtilizationLoss = \frac{C_{i2}}{T_i - C_{i1}} - \frac{C_{i2}}{T_i} \ge 0$. Since higher utilization loss causes lower total utilization of system, when we are forced to split a task, a whole task with the least utilization loss is selected.

IV. Overrun-Freeness Verification of RMLS

In this section, we assume that two processors p_k and p_{k+1} share a task $\tau_i = (T_i, C_i)$ and for each request of the common task C_{i1} is executed by p_k and C_{i2} is executed by p_{k+1} such that $C_i=C_{i1}+C_{i2}$.

Lemma 1: *If Liu&Layland's bound is satisfied by all processors, the second part of a request from a shared task, τ_i, between two processors, p_k and p_{k+1}, never overruns.*

Proof: The preference of executing a request from a shared task τ_i between processors p_k and p_{k+1} is always given to p_k. Whenever p_k is not executing such a request p_{k+1} will be executing it unless the execution of the second part of the request is completed. This is because this request has the highest priority in p_{k+1}. Therefore, in the worst case, the execution of the second part of the task will be complete after a time length of C_i is passed since the request is received, where $C_i \leq T_i$.■

Definition 1: a *conflict-idle* period is a time interval in which both processors, p_k and p_{k+1}, that share the shared task, τ_i, want to run a request from the task but because p_k is given a higher precedence it will proceed with the execution; and at the same time, there is no other pending request for processor p_{k+1} within this period and it will be idle. Note that, not all conflict periods of processors p_k and p_{k+1} are necessarily conflict-idle because if there are other requests for p_{k+1} it will proceed with their execution and hence it will not be idle.

Lemma 2: *If the utilization of each of the two processors, p_k and p_{k+1}, which share a tasks, τ_i, is not higher than Liu and Layland's bound and there is no conflict-idle period with respect to the share task, both processors always run their corresponding tasks safely.*

Proof: Since processor p_k has a higher precedence to run the shared task τ_i than p_{k+1}, this processor will always run safe. On the other hand, the only effect that p_k can have on tasks of processor p_{k+1} is that it may cause the execution of the second part of a request from the shared task to be postponed. This may harm the safety of the shared task in p_{k+1} but it may be beneficial to other tasks of this processor. However, in Lemma 1 it is proven that the second part of a request from a shared task never overruns.■

Lemmas 1 and 2 will hold even if actual utilization of subtask τ_{i2}, i.e. $\frac{C_{i2}}{T_i}$, is used in the computation of utilization of p_{k+1}. It is for compensation of possible conflict-idle periods that, in general, effective utilization of the shared task on processor p_{k+1} is computed as $\frac{C_{i2}}{T_i - C_{i1}}$.

Definition 2: *Effective utilization of a request* (not a task or subtask) at a given time t is defined as below:

$$E_{\tau_i,t} = \frac{Remaining\ execution\ time\ of\ \tau_i}{Remaining\ time\ to\ deadline\ for\ \tau_i}$$

For example, suppose task $\tau = (10, 4)$ has generated a request at time 20 and current time is 26 and up to now this request has received 1.5 unit of CPU time then the effective utilization of the request at time 26 is (4-1.5)/(30-26)=0.625.

Lemma 3: *Suppose two processors p_k and p_{k+1} share a task τ_i. Effective utilization of a request from τ_i for processor p_{k+1} is maximal at the exact time when the execution of processor p_k's share of this request is completed and p_k starts this request immediately after it is generated and continues until completion.*

Proof: Suppose as soon as a request from τ_i is generated at a time t_0 processor p_k starts executing it until its share is finished at time t_0+C_{i1}. At this time effective utilization of the subtask τ_{i2} on p_{k+1} is equal to $\frac{C_{i2}}{T_i-C_{i1}}$. We show that this is in fact maximal effective utilization of τ_{i2}, which means subtask τ_{i2}'s effective utilization never becomes greater than this. Recall that requests of task τ_i have the highest priority in processor p_{k+1}. This implies that any request from this task will be immediately pick up for execution by p_{k+1} if p_k is not executing it. On the other hand, if the execution of the second part of a request from task τ_i is completed by processor p_{k+1}, then its effective utilization becomes zero and remains zero until a new request is generated from the same task. With these points in mind, consider a situation where at any time t_1, $t_0 \leq t_1 \leq t_0+C_i$, processor p_k has executed this request for duration of length a, $a \leq C_{i1}$, and processor p_{k+1} has executed the same request for duration b, $b < C_{i2}$ and $a+b = t_1-t_0$. See Figure 1.

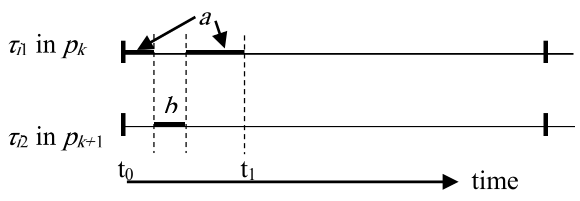

Fig. 1. A Sapmle execution of parts of a split task.

At time t_1 effective utilization of τ_{i2} is $\frac{C_{i2}-b}{T_i-(a+b)}$.

Since $a \leq C_{i1}$,

$$\frac{C_{i2}-b}{T_i-(a+b)} \leq \frac{C_{i2}-b}{T_i-(C_{i1}+b)} = \frac{C_{i2}-b}{T_i-C_{i1}-b}$$

To show that maximal effective utilization of τ_{i2} is $\frac{C_{i2}}{T_i-C_{i1}}$ it has to be shown that $\frac{C_{i2}-b}{T_i-C_{i1}-b} \leq \frac{C_{i2}}{T_i-C_{i1}}$.

That is, $(C_{i2}-b)(T_i-C_{i1}) \leq C_{i2}(T_i-C_{i1}-b)$

Or,

$$-bT_i + bC_{i1} \leq -bC_{i2}$$

Or,

$$b(C_{i1}+C_{i2}) \leq bT_i$$

which is always true because b is positive and $C_{i1}+C_{i2} \leq T_i$.

Theorem 1: *If effective utilization of each of two processors p_k and p_{k+1} which share a task τ_i, is not greater than Liu and Layland's bound, both processors will always safely run their corresponding tasks.*

Proof: This theorem is similar to Lemma 2 in which it is assumed that there will be no conflict-idle period. However, here, this restriction is removed. In Lemma 2, it is mentioned that processor p_{k+1} does not have any influence on the execution of tasks and subtasks assigned to processor p_k. Since

Liu and Layland's bound is satisfied for p_k it will always safely run its assigned tasks. In the packing algorithm, the utilization of the shared task on processor p_{k+1} is computed as $\frac{C_{i2}}{T_i - C_{i1}}$ which, based on Lemma 3, is the maximum utilization which τ_{i2} can ever impose on the processor. On the other hand, the utilization is taken to be less than or equal Liu and Layland's bound. Therefore, this processor will always safely run its assigned tasks, too.■

V. Simulations

In this section, the proposed method is compared with SPA2. We used UUnifast algorithm [18] to produce random unbiased task-sets in which each task's utilization must not exceed one. For each category of task sets, e.g., task sets with total utilization equal to 4, the total of 3000 task-sets, with different number of tasks are generated. For RLMS we do not have to know the number of cores in advanced but we must know it for SPA2. Therefore, for a fair comparison, for SPA2 and for each tasks set, we had to find the overrun-free case with the least number of processors. As the minimum number of processors needed for each method are found, the average utilization of all processor is calculated by dividing overall utilization of the task-set by the number of processors used.

To be brief, only two experiments are shown here. In the first experiment, for task-sets with total utilization equal to 16 and task sets of sizes 38, 48, 67, 106, and 183, the calculated average utilizations are depicted in Figure 2. RLMS leads to an average utilization which is always higher than that of SPA2. Figure 3 shows number of cores used by each method.

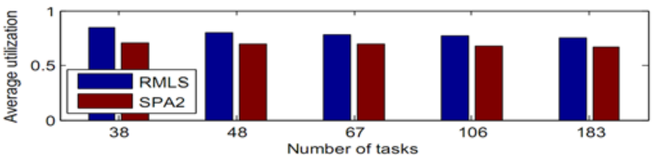

Fig. 2. Average of performance, by each method, for U=16

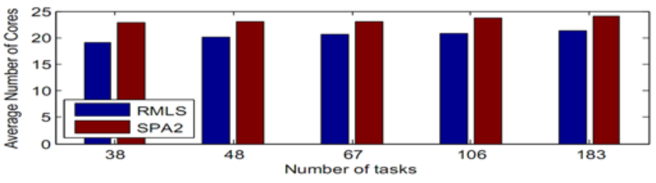

Fig. 3. Number of cores used for each method, for U=16

In the second experiment, rates of schedulable tasks are compared. Figure 4 shows the result of one such experiment where average utilization of task sets grows from 0.5 to 1.0.

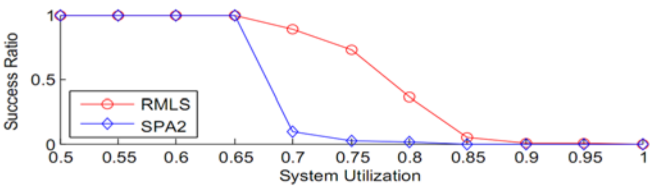

Fig. 4. Rate of schedulable task-sets

More experiments should be performed on RMLS and also should be compared with other methods. Finding a utilization bound for RMLS is in progress.

References

[1] C.L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment". *JPL Space Programs Summary*, vol. 37-60, pp. 28–31, 1969.

[2] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*. IEEE, 2009, pp. 239–248.

[3] M. R. Gary and D. S. Johnson: "Computers and Intractability; A Guide to the Theory of NP-Completeness" (W. H. Freeman & Co.), 1979

[4] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with liu and layland's utilization bound," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 165–174.

[5] J. Anderson, V. Bud, and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 2005, pp. 199–208.

[6] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in Embedded and Real-Time Computing Systems and Applications, *2006. Proceedings. 12th IEEE International Conference on*. IEEE, 2006, pp. 322–334.

[7] A. Burns, R. I. Davis, P. Wang, and F. Zhang, "Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme," *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, 2012.

[8] S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 139–148.

[9] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*. IEEE, 2009, pp. 249–258.

[10] Bletsas, K. & Andersson, B. "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound" *Real-Time Systems*, vol. 47, no. 4, pp. 319-355, 2011

[11] N. Guan and W. Yi, "Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2470–2473.

[12] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[13] S. Kato and N. Yamasaki, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 23–32.

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

[15] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[16] M. Naghibzadeh, P. Neamatollahi, R. Ramezani, A. Rezaeian, and T. Dehghani, "Efficient semi-partitioning and rate-monotonic scheduling hard real-time tasks on multi-core systems," in *Industrial Embedded Systems (SIES), 2018 8th IEEE International Symposium on*. IEEE, 2013, pp. 85–88.

[17] M. Naghibzadeh, and K.H. Kim "The yielding-first rate-monotonic scheduling approach and its efficiency assessment", *International Journal of Computer System Science & Engineering*, 2003, pp. 173-180

[18] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

Phase-aware Scratchpad Memory Management Technique for Saving Energy of Embedded Systems

Chia-Chung Lee, Chen-Wei Huang and Shiao-Li Tsao
Dept. of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
david2658@gmail.com, cwhuang.cs96g@nctu.edu.tw, sltsao@cs.nctu.edu.tw

Abstract—**Embedded applications such as multimedia usually have better memory access predictability thus providing the chance to utilize scratchpad memory (SPM) as a better alternative to the hardware cache in terms of performance and energy [2][6]. Static allocation techniques lock the most frequently executed parts in the SPM, do not alter the content, and are simple and effective approaches. However, for complex applications with numerous hotspots, static allocation techniques usually lead to underutilization of the SPM space. Dynamic SPM allocation techniques remedy the shortcoming by allowing the freedom of altering the SPM composition at runtime. However, for large real applications such as multimedia codecs exhibiting clear phase behavior [15], the importance of the code and data object may vary significantly in different execution phases indicating inflexibility of fixed classification and lower SPM utilization. Therefore, in this study, we incorporate phase detection techniques in dynamic SPM management. Preliminary results demonstrate the phased-aware SPM management outperforms the conventional dynamic SPM schemes.**

Keywords—scratchpad memory; phase behavior; dynamic management

I. Introduction

Reducing energy consumption is important for battery-operated devices such as handsets and tablets. Hardware-controlled caches pull the gap between external memory and CPU at runtime transparently to the program at the cost of auxiliary hardware control logic, however limited by the visibility scope, in an ad-hoc manner. In contrast, software-controlled caches (also known as Scratchpad Memory, SPM) stripped the area and energy premium of hardware control units at the cost of requiring delicate software analysis at design time. For applications with better memory access predictability such as multimedia in the embedded systems, SPM can serve as a better alternative than the hardware cache in terms of performance and energy [2] [6].

Scratchpad memory (SPM) is widely used in embedded CPUs such as ARMv6 cores [9], IBM Cell SPU [6] and in recent Nvidia [13] and AMD GPUs [14]. Abundant research have been devoted to effectively use the SPM as a placeholder for data and/or code either statically [11] [12] or dynamically [1] [2] [3] [4] [6] [9]. Both approaches classify the code and/or data objects to be accessed from the SPM or main memory. Static allocation techniques lock the most frequently executed parts to the SPM and do not alter the content. For simple applications with a few hotspots, static allocation is a simple and effective approach. However, for complex applications with numerous hotspots, static allocation techniques usually lead to underutilization of the SPM space. Dynamic SPM allocation techniques remedy the shortcoming by allowing the freedom of altering the SPM composition at runtime.

Depending on the way that the features of code and data objects are gathered, dynamic SPM management schemes can be broadly categorized into two classes, namely profile-based [1] [2] [3] and static-based analysis [4] [6]. Profile-based methods obtain the information via executing the target program several times with different inputs to synthesis a representative executed trace which classification depends. Static-based analysis analyzes the control flow graph of each function together with call graph representing possible caller-callee relationship to derive possible interferences between functions without actually executing the program. Both approaches classify the code and data objects into fixed categories, that is either they will be accessed from the SPM or main memory whenever needed during the entire execution.

However, for large real applications such as multimedia codecs exhibiting clear phase behavior [15], the importance of the code and data object may vary significantly in different execution phases indicating inflexibility of fixed classification and possibly lower SPM utilization. Experimental results [1] show that the state-of-the-art approaches do not adequately handle programs with many phases such as the H.264 decoder.

Program phase behavior has been exploited to improve performance or power consumption in hardware reconfiguration and dynamic translation [5]. To the best of our knowledge, this is the first study to incorporate phase detection techniques in dynamic SPM management. Since memory transfer is expensive, we will also consider the possibility of retaining a code object classification across two consecutive phases. Considering the inter-phase correlation makes our approach unique from previous phase-aware optimizations.

The rest of this paper is organized as follows: Section II discusses previous work in the dynamic SPM allocation techniques and program phase detection. In Section III, we give an educational example to demonstrate the benefit of incorporating phase concept to better illustrate our main ideas. Section IV presents the preliminary result of phase detection and phased-aware SPM management. Section V discusses our future work.

II. Related Work

Fruitful work concerning optimal use of scratchpad memory exist. We refrain the discussion here to studies which are more closely to our focus in dynamic SPM management of code objects. For broader discussions, please refer to [12] [9] [11]. Previous studies [1] [2] [3] [4] [6] presented various approaches to decide management strategies with the goals of reducing energy consumption. Most of the studies [1] [2] [3] are profile-based approaches, and comparatively less studies [4] [6] are based on purely static analysis techniques.

Egger et al. [1] classified the code into three groups (pinned, paged, and external). Pinned code objects are loaded and remain in the SPM. Paged code objects are loaded into the SPM from the main memory on demand during execution with a simple LRU replacement. External code objects are always fetched directly from the external memory when required. A SPM manager is in charge to manage the dynamic update of the SPM content. They formulated an integer liner programming (ILP) model to classify the code to minimize the energy consumption. In [2], authors presented an architecture consisting of the SPM, mini-cache, and MMU. The MMU converts virtual address to physical address, and uses a comparator to lead the access to either the SPM or mini-cache. A page manager is responsible to update the page table enabling the MMU to detect a page miss in the SPM. The page manager updates the SPM content when receiving a page fault from the MMU. In [3], Verma et al. introduced a two-step algorithm determining the best set of memory objects to be accessed from the SPM and the optimal spill locations to load and unload them from and to the main memory.

Baker et al. [4] proposed a greedy algorithm to generate an overlay strategy to minimize interferences between functions mapping to the same region with purely static analysis of the code structure. Pabalkar et al. [6] expanded the function nodes in the call graph to their respective control flow graphs to create a data structure called Global Call Control Flow Graph (GCCFG). Weights are assigned to nodes to calculate the interferences between functions mapping to the same SPM region. They presented an ILP model and heuristic to calculate the functions mapping to the overlay regions.

Dhodapkar and Smith [5] compared three program detection techniques, namely working set signatures based, basic block vectors based, and conditional branch counter based. The working set signatures approach compare two consecutive working sets using a similarity metric. Phase change is indicated when the metric exceeds the preset threshold. The basic block vectors (BBVs) record the basic block executed frequency during a particular fixed-length execution interval. It compares two consecutive BBVs using Manhattan distance. When the distances exceed a threshold, it indicates a program phase change. The conditional branch counter approach tracks the count of conditional branch during a fixed-length interval. When the difference in branch counts of two consecutive intervals exceeds a threshold, program phase change is detected. In their conclusions, the BBV based technique provides better sensitivity and lower performance variation in phases compared to other techniques. On the other hand, instruction working set technique provides slightly higher stability and longer phases length.

III. Main Idea of Phase-Aware Allocation Techniques

We use an educational example to illustrate our idea. The example program has four functions as shown in Fig. 1, and the function B is the program entry. We execute the program and collect the function trace as follows (exponent means repetitions):

$$((BA)^5BD)^6(BA)^2((BC)^2BD)^5.$$

Table I summarizes the size, executed time percentage, and the total called count of each function. Table II shows the percentage of time that functions executed in each time slot. In the following subsections, we first allocate the SPM without using the concept of program phases in Section III.A. Then we demonstrate the benefits of incorporating the concept of program phases in Section III.B. Finally, we compare two approaches in energy consumption in Section III.C.

```
Function A
    // Do something
End A

Function B
    For i from 1 to 32 Do
      Call A
      If i mod 5 is 0 Then Call D
    End for
    For j form 1 to 10 Do
      Call C
      If j mod 2 is 0 Then Call D
    End for
End B

Function C
    // Do something
End C

Function D
    // Do something
End D
```

Fig. 1. Educational example.

TABLE I. Functions metadata.

Func.	Information of the function		
	Function size	*Total executed time percentage*	*Total call count*
A	128 bytes	70%	32
B	64 bytes	6.25%	1
C	64 bytes	17.5%	10
D	64 bytes	6.25%	11

TABLE II. Executed time percentage of each time slot.

Func.	Percentage of each time slot							
	S1	*S2*	*S3*	*S4*	*S5*	*S6*	*S7*	*S8*
A	90%	90%	90%	90%	90%	90%	20%	0%
B	5%	5%	5%	5%	5%	5%	12%	8%
C	0%	0%	0%	0%	0%	0%	60%	80%
D	5%	5%	5%	5%	5%	5%	8%	12%

TABLE III. Phase composition.

Phase name	Phase composition

Phase name	Phase composition
A	Reading network abstraction layer units
B	Entropy decoding and motion compensation
C	Deblocking filter

A. ***Approch A**: without the concept of program phase*

According to Table I, we may think function A is the hottest function, with function C being the second, and function B and D being the least important ones. Assume that there is a 192-byte SPM partitioned into three pages each with a fixed size of 64 bytes. Using the ILP model proposed by Egger et al in [1], function A will be classified as a pinned function occupying 2 pages of the SPM without replacement during the whole execution. Function B, C, and D will compete for the remaining 1-page free SPM space as demanded. All works well during time slots S1 to S6. Function B and D replace each other in turn. However, during time slot S7 and S8, function C is actively used, in contrast function A ceases to appear but occupying valuable SPM resource. This leads to the unpleasant under-utilization of SPM resource by having functions B and C repeatedly evicting each other. This situation results in consuming large amount of energy for copying functions into the SPM from the external memory which downgrades the performance.

B. ***Approch B**: with the concept of program phase*

We add the concept of program phase to this example. According to Table II, using a simple function vector collecting the frequency of each function called during the time slot as the metric, we can detect there is a phase change during time slot S6 and S7. We divide the original trace into two sub-traces corresponding to phase transition point.

$$((BA)^5BD)^6, \text{ and } (BA)^2((BC)^2BD)^5.$$

The first part consists of the sub-trace in time slot S1 to S6, and the second part corresponds to time slot S7 to S8. Using the same ILP model in [1] twice on the two sub-traces gives us two allocation schemes. The solution of first part classifies function A as a pinned function, whereas function B and D classified as paged functions, and function C is classified as the external function which is fetched directly from the external main memory bypassing the SPM. The solution is similar to *Approach A* in the SPM space distribution. The difference between *Approach A* and *Approach B* is the role of the function C. However, the program does not call function C in S1-6. The solution corresponding to the second phase classifies function A, C, and D as paged functions, and classifies function B as a pinned function. Function B is loaded in time slot S7 and remains in the SPM during S7-8. Other three functions are moved into the SPM dynamically as demanded. Unlike the fixed allocation scheme mentioned in Section III.A, we change the allocation scheme dynamically tailored for the current program phase.

C. *Compare two approaches*

Assume that fetching an instruction from the SPM and fetching an instruction from the external memory cost 1-unit, and 10-unit energy, respectively.

As the function misses in the SPM, there is the miss penalty including the function relocation and management energy consumption spending. The management composes of 100 instructions in the SPM, and it costs 100-unit energy. As a new phase occurs, the manager needs to swap in the suitable management strategy for this phase and initialize the SPM state. This is more complex than management and we assume it costs 1000-unit energy.

In *Approach A*, the manager is called for moving functions 43 times, and functions B, C, and D are moved into the SPM 22 times, ten times, and eleven times, respectively. At loading time, the manager loads function A. The total management energy consumption is 31820-unit energy.

In *Approach B*, the manager is called for moving functions 13 times during S1-6 and moving functions three times during S7-8. Function B and C are moved into the SPM seven times and six times, respectively during S1-6, and function A, C, and D are moved into the SPM only once, respectively. At the starting of S1, the manager loads the function A, and as the program phase change, the manager loads the function B into the SPM. The total management energy consumption is 13480-unit energy. Using the concept of program phase, we can reduce the management overhead by 58%.

IV. Preliminary results

In this section, we use the H.264 reference decoder [7] as the target benchmark compiled on ARM platform with several video clips. To have a quick estimation on the benefits of exploiting phase behavior in dynamic SPM management, we use the intuitive approach of dividing the program phases from an algorithmic point of view. Table III lists the major tasks in each phase. In the following subsections, we first verify the quality of dividing phases using an algorithmic approach with the optimal management strategy in Section IV.A. Then we point out the benefits of incorporating the concept of program phases in Section IV.B.

A. *OPT and Phase-OPT management*

Assume the CPU is equipped with a 32KB instruction SPM, and functions are divided into 128-byte pages and aligned to the page size. In a paged environment with a fixed number of available pages, the optimal page-replacement algorithm (*OPT*) is known to have the lowest page-fault rate [8]. To evaluate the efficiency of our algorithmic-based phase partition, we use *OPT* as the page replacement strategy in both phased and the original non-phased trace to understand the extra page-fault penalty. In a phased trace, we partition the original non-phased trace by phase entry points. In contrast to the phased trace where *OPT* has complete knowledge about the future, in a phased-trace the *OPT* has no visibility beyond the phase boundary when determining which page to evict. To be conservative, we assume the worst case that the SPM is completely flushed during phase transition.

The comparison result is quite promising. For the original non-phased trace, *OPT* moves 23,404,928 bytes from the external memory to the SPM, and for the phased-trace *OPT* moves 24,502,912 bytes. The increasing penalty is merely 4.69%. The increment of copying is caused by flushing contents in the SPM while phase changing and the wrong decisions of replacement. The small extra page-fault penalty indicates that we have found a good boundary where the accesses beyond the boundary are irrelevant to the replacement choices.

B. SPM allocation for H.264 decoder

We first collect instruction traces for constructing the ILP formulation, and use one of the state-of-the-art dynamic SPM management approach [1] for classifying the functions in both phase-aware and phase-unaware manner. Table IV lists classification of functions in both original phase-unaware and proposed phase-aware manner. The first column of the Table IV is the original approach, and the other three columns is the phase-aware approach.

In the original phase-unaware strategy, the classification of function remains fixed. In comparison, the classification of functions adapts to the need of each phase in our proposed phase-aware strategy. The simulation results are illustrated in Table V. We estimate the energy consumption using (1) with the following parameter definitions.

$$\text{Energy} = (A_{pinned} + A_{paged}) \times E_{SPM_r} + A_{ext} \times E_{ext} + M \times E_{manage} + C \times (E_{ext} + E_{SPM_w}) + P \times E_{change} \quad \text{... (1)}$$

A_{pinned}	Accessed bytes from pinned functions;
A_{paged}	Accessed bytes from paged functions;
A_{ext}	Accessed bytes from external functions;
M	Number of byres moved by the manager;
C	Nomber of times which paged function misses;
P	Number of times which the phase changes;
E_{SPM_r}	Energy of reading one byte from SPM;
E_{SPM_w}	Energy of writing one byte from SPM;
E_{ext}	Energy of reading one byte from external memory;
E_{manage}	Energy of one paged function missing;
E_{change}	Energy of one phase changing.

Using CACTI [16], we can obtain the energy consumption value for the E_{SPM_r}, E_{SPM_w}, E_{ext}, E_{manage}, and E_{change} is 0.210442, 0.00361979, 4.08893, 21.0442, and 6752.73 (nj), respectively. The original approach consumes 5,007,430,307.85 (nj), and phase-aware approach consumes 752,307,327.20 (nj). The phase-aware effectively reduces the energy consumption by 85%. We found that many functions in the original phase-unaware approach are classified into paged functions whereas they were promoted to pinned class in the phase-aware classification. This effectively reduces the competition of the limited buffer for paged functions in the SPM.

V. Future work

In the future, we will propose a dynamic SPM management strategy with suitable phase identification techniques. We will also develop a cost model incorporating the inter-phase behavior.

References

[1] Bernhard Egger, Seungkyun Kim, Choonki Jang, Jaejin Lee, Sang Lyul Min, and Heonshik Shin. Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU. In IEEE Transactions on Computers, Vol. 59, No. 8, 1047-1062, August 2010.

[2] Bernhard Egger, Jaejin Lee and Heonshik Shin. Dynamic Scratchpad Memory Management for Code in Portable Systems with an MMU. ACM Transactions on Embedded Computing Systems, Vol. 7, No. 2, Article 11, February 2008.

[3] Manish Verma, and Peter Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 8, 802-815 Auguest 2006.

[4] Michael A. Baker, Amrit Panda, Nikhil Ghadge, Aniruddha Kadne, and Karam S. Chatha. A Performance Model and Code Overlay Generator for Scratphad Enhanced Embedded Processors. CODES+ISSS'10, October 24-29, 2010, Scottsdale, Arizona, USA, 2010.

[5] Ashutosh S. Dhodapkar and James E. Smith. Comparing Program Phase Detection Techniques. Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03), 2003.

[6] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous Determination of Regions and Function to Region Mapping for Scratchpad Memories. HiPC 2008, LNCS 5374, p. 569-582, 2008.

[7] H.264/AVC JM Reference Software, http://iphome.hhi.de/suehring/tml/

[8] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating Systems Concepts (Eighth Edition). Wiley 2010, p. 374-376.

[9] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. DAC 2001,June 18-22, 2001, Las Vegas, Nevada, USA, 2001.

[10] ARM1156T2-S™ Technical Reference Manual. http://infocenter.arm.com/. 2007.

[11] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. EDTC '97 Proceedings of the 1997 European conference on Design and Test Page 7, 1997.

[12] Chen-Wei Huang and Shiao-Li Tsao. Minimizing Energy Consumption of Embedded Systems via Optimal Code Layout. Computers, IEEE Transactions on (Volume:61 , Issue: 8), p. 1127-1139, Aug. 2012.

[13] CUDA Toolkit Documentation v6.0. http://docs.nvidia.com/cuda, Developer Zone, 2014.

[14] Bryan Catanzaro. OpenCL™ Optimization Case Study: Support Vector Machine Training123. http://developer.amd.com/, Developer Center 2011.

[15] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. Micro, IEEE (Volume:23 , Issue: 6), p. 84-93, Nov.-Dec. 2003.

[16] Cacti 6.5. http://www.hpl.hp.com/research/cacti/, HP Labs.

TABLE IV. Number of classified functions.

Classification	Number of functions			
	Original	Phase A	Phase B	Phase C
pinned	35	56	54	42
paged	132	8	0	8
external	210	313	323	327

TABLE V. Result comparison.

	Original	Phase-aware
pinned	708,197,476 bytes	1,265,710,060 bytes
paged	543,316,108 bytes	49,577,936 bytes
external	179,532,968 bytes	115,758,556 bytes
Managing count	4,026,953*	208*
Copy functions	959,112,960 bytes	84,480 bytes
Phase changing count		272*

* are the number of management of function missing, and the number of management of phase changing, respectively.

Towards Holistic Analysis for Fork-Join Parallel/Distributed Real-Time Tasks

Ricardo Garibay-Martínez[1], Luis Lino Ferreira[1], Geoffrey Nelissen[1], Paulo Pedreiras[2], Luís Miguel Pinho[1]
[1]CISTER/INESC-TEC, ISEP, Porto, Portugal
[2]DETI/IT/University of Aveiro, Aveiro, Portugal
[1]{rgmaz, llf, grrpn, lmp}@isep.ipp.pt; [2]pbrp@ua.pt

***Abstract*—Parallel/distributed processing is a solution for providing scaling computing power for computational-intensive applications. Parallel/distributed applications are commonly based on the fork-join model, where precedence constraints have to be considered on the development of an adequate timing analysis. Moreover, as the main difference with multicore architectures, distributed systems entail the transfer of messages upon a communication network that should be integrated in the timing analysis. In this context, this paper presents the current status of the work towards holistic analysis for fixed priority fork-join parallel/distributed tasks. This analysis takes into consideration the interactions between parallel threads and their respective messages. These considerations will be helpful for the improvement of the combination of existing results for computing the worst-case response time and the specific case of fork-join parallel/distributed real-time tasks.**

Keywords—Real-time; parallel execution; distributed systems; holistic analysis.

I. Introduction

Modern real-time applications are increasingly complex requiring the use of more powerful computing resources. The current trend of using parallel processing in the embedded domain seems a promising solution to cope with the requirements of such demanding applications. Therefore, the real-time community has been making efforts to extend traditional real-time tools and scheduling algorithms to consider parallel task models. However, in some embedded applications, the use of powerful enough multi-core processors, is prohibited due to Size, Weight, and Power (SWaP) constraints. But it is also possible to comply with the requirements of computational-intensive applications by allowing single-core embedded devices connected through a local real-time network, to distribute its workload to remote neighbour nodes and execute the applications in parallel.

In this work we consider fork-join distributed real-time applications [1] which are composed of a set of fork-join Parallel/Distributed real-time tasks (P/D tasks), executing in a distributed system. When considering such tasks, the processing of tasks and messages must comply with their associated time constraints. A P/D task starts by a master thread executing sequentially; and then forks to be executed in parallel on *remote processors*. When the parallel execution has completed on each of the remote processors, the results are aggregated by performing a join operation and the execution of the sequential thread is resumed within the master thread. We call to those operations, the Distributed-Fork (D-Fork) and Distributed-Join (D-Join).

We also consider that P/D tasks are scheduled with the *Partitioned/Distributed - Deadline Monotonic Scheduling* (P/D-DMS) algorithm, proposed in [2]. The P/D-DMS algorithm can be used for partitioning a set of threads onto uniprocessor nodes connected through a real-time network. The algorithm makes use of the *Distributed Stretch Transformation* (DST) [2]. After applying the DST, a set of P/D threads have to be assigned onto the nodes of the distributed systems, which is done by using the Fisher Baruah Baker - First Fit Decreasing (FBB-FFD) algorithm [3].

Goal of this work. In a previous work, Axer *et al.* [4] presented a method for computing the response time of fixed-priority parallel tasks on multiprocessors, which considers the synchronization effects of fork-join tasks. In this paper, we extend the existing holistic analysis for the computation of the Worst-Case Response Time (WCRT) of sequential tasks in a distributed system, to *parallel* distributed (P/D) tasks. When considering P/D tasks, the transmission delays due to the messages exchanged by communicating threads within a P/D task cannot be considered negligible as it is the case on multiprocessor platforms. Furthermore, we consider the specific structure of the P/D tasks after applying the P/D-DMS algorithm, and its impact when computing their WCRT.

II. System Model

Formally, we consider that a distributed real-time application is composed of a set $\tau = \{\tau_1, \dots, \tau_n\}$ of n *P/D tasks* [2]. Figure 1 shows an example of a P/D task τ_i. A P/D task τ_i is activated periodically every T_i time units and is characterised by an implicit end-to-end deadline D_i. Also, it is considered that all P/D tasks are released synchronously. A P/D task τ_i ($i \in \{1, \dots, n\}$) is composed of a sequence of sequential and parallel/distributed (P/D) segments $\sigma_{i,j}$ with $j \in \{1, \dots, n_i\}$. Where, n_i represents the number of segments composing τ_i, n_i is assumed to be an *odd* integer, as a P/D task should always start and finish with a sequential segment. Therefore, odd segments $\sigma_{i,2j+1}$ identify sequential segments and *even* segments $\sigma_{i,2j}$ identify P/D segments. Each segment $\sigma_{i,j}$ is composed of a set θ of threads $\theta_{i,j,k}$ with $k \in \{1, \dots, n_{i,j}\}$, where $n_{i,j} = 1$ for sequential segments and $n_{i,j} = m_i \leq m$ threads for P/D segments. m_i is the number of P/D threads in each P/D segment, and it is considered to be the same for all P/D segments within a P/D task τ_i. m is the number of distributed nodes.

All sequential segments within a P/D task τ_i must execute within the same processor. This means that the processor that performs a D-Fork operation (*invoker processor*) is in charge of aggregating the result by performing a D-Join operation. Threads within a P/D segment are possibly executed on remote processors. Consequently, for each thread $\theta_{i,2j,k}$ belonging to a P/D segment (P/D thread), two P/D messages $\mu_{i,2j-1,k}$ and $\mu_{i,2j,k}$ are considered for realizing the communication between the invoker and remote processors. That is, P/D threads and messages that belong to a P/D segment and execute on a remote processor, have a precedence relation: $\mu_{i,2j-1,k} \longrightarrow \theta_{i,2j,k} \longrightarrow \mu_{i,2j,k}$. For each sequential and P/D segment, there exists a synchronisation point at the end of each segment, indicating that no thread that belongs to the segment after the synchronisation point can start executing before all threads of the current segment have completed execution. P/D threads are preemptive, but messages packets are non-preemptive, although large messages can be divided in several non-preemptive packets.

Also, each sequential thread $\theta_{i,2j+1,1}$ has a Worst-Case Execution Time (WCET) of $C_{i,2j+1,1}$. A P/D thread $\theta_{i,2j,k}$ has a WCET of $P_{i,2j,k}$, and each message $\mu_{i,j,k}$ has a Worst-Case Message Length (WCML) $\mathcal{M}_{i,j,k}$. It is assumed that for a task τ_i, every P/D thread $\theta_{i,2j,k}$ and their respective messages $\mu_{i,j,k}$ within a P/D segment $\sigma_{i,2j}$, have identical WCETs $P_{i,2j,k}$ and identical WCMLs $\mathcal{M}_{i,j,k}$, respectively. However, the WCET and the WCML of P/D threads and their messages can vary between different P/D segments. Also, P/D threads and P/D messages within a task τ_i, share the same period T_i.

To summarise, it is possible to describe a P/D task as:

$$\tau_i = ((C_{i,1}, \mathcal{M}_{i,1}, P_{i,2}, \mathcal{M}_{i,2}, C_{i,3}, \dots, \mathcal{M}_{i,n_i-1}, C_{i,n_i}), m_i, T_i),$$

where:

- n_i is the total number of segments of a task τ_i,
- $C_{i,j}$ is the WCET of each sequential segment $\sigma_{i,2j+1}$,
- $\mathcal{M}_{i,j}$ is the WCML of a single messages (all m_i P/D messages on the same P/D segment have the same WCML),
- $P_{i,2j}$ is the WCET of a single P/D thread within a segment $\sigma_{i,2j}$ (all m_i P/D threads on the same P/D segment have exactly the same WCET),
- m_i is the number of P/D threads (two messages are created for each P/D thread within a P/D segment $\sigma_{i,2j}$) in each P/D segment,
- T_i is the period of a task, which is equal to its deadline ($D_i = T_i$).

A. Preliminaries

For notational convenience we introduce some definitions that will simplify the explanation of the P/D-DMS [2] algorithm.

Definition 1. (Master thread). *The master thread of a P/D task τ_i is the collection of all threads $\theta_{i,j,1}$ belonging to all segments $\sigma_{i,j}$. A master thread can be represented as:*

$$\tau_i^{master} = \{\theta_{i,1,1}, \theta_{i,2,1}, \theta_{i,3,1}, \dots, \theta_{i,n_i-1,1}, \theta_{i,n_i1}\} \quad (1)$$

Definition 2. (Minimum execution length). *The minimum execution length η_i represents the minimum execution time a P/D task τ_i needs to execute, if all P/D threads are executed in parallel. This is equal to the sum of the WCET of all the threads described in the master thread:*

$$\eta_i = \left(\sum_{j=0}^{\frac{n_i-1}{2}} C_{i,2j+1}\right) + \sum_{j=1}^{\frac{n_i-1}{2}} P_{i,2j,1} \quad (2)$$

Definition 3. (Maximum execution length). *The maximum execution length C_i, represent the maximum execution time a P/D task τ_i needs to execute when all P/D threads are executed sequentially on the invoker processor. This is equal to the sum of WCET of all threads in a task τ_i:*

$$C_i = \left(\sum_{j=0}^{\frac{n_i-1}{2}} C_{i,2j+1}\right) + \left(\sum_{j=1}^{\frac{n_i-1}{2}} P_{i,2j,1}\right) \times m_i \quad (3)$$

Definition 4. (Slack time). *The positive slack time L_i is the temporal difference between the task's deadline D_i and the minimum execution length η_i:*

$$L_i = D_i - \eta_i \quad (4)$$

If the slack L_i is a negative number, it means that η_i is larger than its deadline ($T_i = D_i$). Therefore, such a task is not schedulable on any number of processors with a speed of 1.

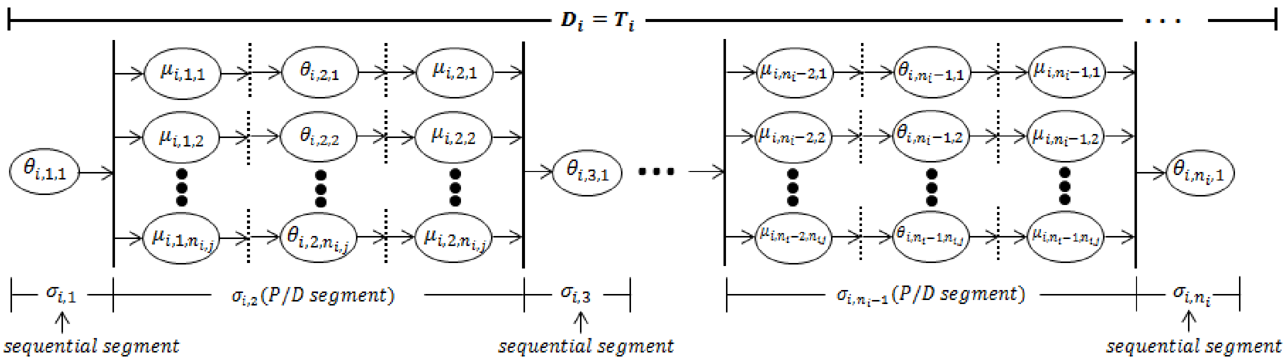

Fig. 1. The fork-join parallel/distributed periodic real-time tasks (P/D task) model.

Definition 5. (Task Capacity). *The task capacity f_i is defined as the capacity of the master thread of a task τ_i to execute extra P/D threads from all P/D segments without missing its deadline:*

$$f_i = \frac{L_i}{\sum_{j=1}^{\frac{n_i-1}{2}} P_{i,2j,1}} \tag{5}$$

III. The P/D-DMS Algorithm

P/D-DMS algorithm [2] is a dispatching algorithm for partitioning a set τ of P/D tasks τ_i onto the elements of the distributed system. The P/D-DMS algorithm realizes the dispatching by: (i) applying the DST [2] to each P/D task τ_i in τ, and (ii) partitioning the set of remaining P/D threads after applying the DST onto processors according to the FBB-FFD algorithm [3]. P/D messages $\{\mu_{i,j,k}^{cd}\}$ are scheduled according to the fixed priority scheduling policy of the network.

The DST was inspired by the SST transformation model [5]. The DST also opts for the formation of a stretched master thread $\tau_i^{stretched}$ for each P/D task τ_i. However, the DST addresses some specific constraints that are related to distributed systems. For example, when realising a D-Fork operation, it implies that some messages will be transmitted within the network, affecting the execution length of the P/D tasks. Let us illustrate the DST transformation with an example. Consider two tasks: $\tau_1 = ((1, 1,2,1,1), 3,8)$, and $\tau_2 = ((1, 1,3,1,1), 3,10)$ to be scheduled on 3 processors. The DST transformation is illustrated in Figure 2. By calculating the maximum execution length (Definition 3) of tasks τ_1 and τ_2, we obtain $C_1 = 8$ and $C_2 = 11$. Then, by looking at Figure 2, it is possible to observe two cases:

1. $C_i \leq T_i$. This is the case of τ_1 in our example; whenever such a case appears for a task τ_i, the task τ_i is fully stretched into a master thread and handled as a sequential task with execution time equal to C_i, a task period of T_i, and an implicit deadline equal to D_i. Therefore, no messages are generated for transmission on the network.

2. $C_i > T_i$. This is the case of τ_2 in our example; for such tasks, the DST transformation inserts (coalesces) as many P/D threads of τ_i into the master thread as possible. To do so, it is first needed to calculate the available slack and capacity of task τ_i as indicated in Eq. (4) and (5). For τ_2, it gives $L_2 = 10 - 5 = 5$ and, $f_2 = 5/3$. Thus, the number of P/D threads that each P/D segment can fully insert into the master thread without causing τ_i to miss its deadline is given by:

   $$i_{i,2j} = \lfloor f_i \rfloor \tag{7}$$

   In the case of τ_2, $i_{2,2} = \lfloor f_2 \rfloor = 1$. Figure 2 shows that τ_2 executes two P/D threads per P/D segment on the invoker processor rather than only one when considering the non-stretched master thread.

The number $q_{i,2j}$ of the remaining P/D threads that have not been coalesced into the master thread is given by:

$$q_{i,2j} = m_i - i_{i,2j} \tag{8}$$

The slack f_i of task τ_i is equally distributed between all the P/D segments of a P/D task τ_i. Thus, the maximum scheduling length for the subset of P/D threads and their respective P/D messages is determined by defining a set of P/D intermediate deadlines $d_{i,2j}$:

$$d_{i,2j} = (f_i + 1) \times P_{i,2j} \;\; \forall\, 1 \leq j \leq \frac{n_i - 1}{2} \tag{9}$$

Thus, at the end of the DST transformation, a P/D task τ_i will be composed of: (i) a single stretched master thread $\tau_i^{stretched}$, and a set of constrained deadline P/D threads $\{\theta_{i,j,k}^{cd}\}$, and their respective constrained deadline messages $\{\mu_{i,jk}^{cd}\}$; per each P/D segment $\sigma_{i,2j}$, or (ii) a single fully stretched sequential task. The stretched master thread $\tau_i^{stretched}$ is assigned to its own processor. The remaining single fully stretched sequential tasks and P/D threads $\{\tau_i^{cd}\}$ are assigned to processors with the FBB-FFD algorithm [3]. Messages $\{\mu_{i,jk}^{cd}\}$ are assigned to the real-time network and scheduled accordingly.

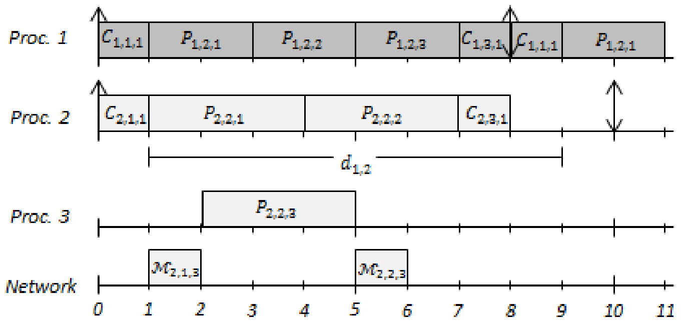

Fig. 2. A task scheduled by the P/D-DMS algorithm.

IV. Holistic Analysis for P/D tasks

The holistic analysis has been conceived as a solution for the analysis of the interaction of a system composed by a set of different processing devices (e.g. processors and networks) [6]. One of the main goals of the holistic analysis approach is to calculate the so called end-to-end delay. The end-to-end delay is the WCRT associated to a chain of tasks executing on the same processor or different processors and interchanging messages for communication purposes. A holistic approach considers the analysis of such a chain of dependencies, which implies higher degree of difficulty when compared to the analysis of components in isolation. However, since the parameters in a holistic analysis are dependent but monotonic, it is possible to formulate a recurrence and progressively iterate until finding a stable solution.

The holistic analysis relies on a simple concept of attribute inheritance, for instance, the activation (release) of a P/D message or P/D thread is based on the response time of previous processing event (e.g. P/D threads or P/D message, respectively). It is possible to observe in Figure 2, that after thread $\theta_{2,1,1}$ has completed execution, the transmission of the message $\mu_{2,1,3}$ is triggered, and in turn this message triggers the execution of the P/D thread $\theta_{2,2,3}$.

Fully stretched tasks: when considering tasks that are fully stretch into a sequential task, no transmissions are required (Case 1, Section III). Therefore, their WCRT only depends on the suffered interference caused by other higher priority threads executing on the same processor.

Non-fully stretched tasks: when considering non-fully stretched tasks, it is necessary to consider the sequential and parallel segments independently. Let us recall that for each sequential and P/D segment, there exist a synchronisation point at the end of each segment, in which threads of the next segment can only continue their execution whenever all threads of the current segment have completed their execution. Therefore, the WCRT of a task τ_i is computed based on the sum of the maximum execution paths of each segment $\sigma_{i,j}$:

$$WCRT(\tau_i) = \sum_{j=1}^{n_i} \max_j(WCRT(\sigma_{i,j})) \tag{10}$$

Sequential segments $\sigma_{i,2j+1}$ within a P/D task, are executed on their own processor, therefore, they do not suffer any interference from other threads. Thus, the maximum WCRT is equal to the WCET ($C_{i,2j+1,1}$) of the corresponding thread $\theta_{i,2j+1,1}$:

$$WCRT\left(\sigma_{i,2j+1}\right) = C_{i,2j+1,1} \tag{11}$$

For parallel segments $\sigma_{i,2j}$ within a P/D task, the maximum WCRT is given by the maximum WCRT of two possible scenarios:

1. the sum of all coalesced P/D threads (denoted as $CThr_{i,2j}$) within the master thread which are executed sequentially:

$$WCRT(CThr_{i,2j}) = \sum_{\forall \theta_{i,2j,k} \in \sigma_{i,2j} \wedge \in master\ thread} P_{i,2j,k} \tag{12}$$

2. or, the $WCRT_{DP_{i,2j}}^{max}$, which is the maximum WCRT of the k distributed execution paths (denoted as $DP_{i,2j,k}$); per each P/D segment. A distributed execution path is the execution of a P/D thread that has not been coalesced with the master thread and their respective P/D messages that have a precedence relation: $\mu_{i,2j-1,k} \longrightarrow \theta_{i,2j,k} \longrightarrow \mu_{i,2j,k}$. For calculating WCRT of a distributed execution path, it is possible to use the following equation:

$$WCRT(DP_{i,2j,k}) = WCRT\left(\mu_{i,2j-1,k}\right) + WCRT\left(\theta_{i,j,k}\right) + WCRT(\mu_{i,2j,k}) \tag{13}$$

Then, it is needed to find the maximum $WCRT(DP_{i,2j,k})$ as:

$$WCRT_{DP_{i,2j}}^{max} = \max_k(WCRT(DP_{i,2j,k})) \tag{14}$$

Thus, for each P/D segment $\sigma_{i,2j}$ within a P/D task, the maximum WCRT is equal to:

$$WCRT\left(\sigma_{i,2j}\right) = \max(WCRT(CThr_{i,2j}), WCRT_{DP_{i,2j}}^{max}) \tag{15}$$

Also, it is important to note that the WCRT of threads and messages depend on the characteristics of the processing elements, and on the particular method to calculate the WCRT. For example, for computing the WCRT of a P/D thread, it is needed to consider the characteristics of the computing nodes (e.g. uniprocessor nodes, multicore nodes, etc.). Likewise for computing the WCRT of P/D messages, it is needed to consider the specific characteristics of the real-time networks (e.g. CAN, FTT-SE, etc.). However, when considering the P/D task model and using a task transformation as the DST, the reasoning of our generic holistic analysis can be used.

V. Conlusions and Future work

In this paper, we presented a generic holistic analysis approach. This analysis studies the specific structure of the P/D tasks after applying the P/D-DMS algorithm, and its impact when computing their WCRT. Although the methods to compute the WCRT of P/D tasks depend on the specific characteristics of computing resources and networks, the holistic analysis reasoning presented in this paper is completely generic. Hence, we are currently working on extracting some characteristics of the P/D task model along with properties of specific communication protocols such as the Flexible Time Triggered protocol, with the intention of improving the computation of the WCRT for parallel tasks in a distributed environment.

Acknowledgments

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER); by FCT and the EU ARTEMIS JU funding, within projects ENCOURAGE (ARTEMIS/0002/2010, JU grant nr. 269354), ARROWHEAD (ARTEMIS/0001/2012, JU grant nr. 332987), CONCERTO (ARTEMIS/0003/2012, JU grant nr. 333053); by FCT and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71562/2010.

References

[1] R. Garibay-Martinez, L. L. Ferreira and L. M. Pinho, "A framework for the development of parallel and distributed real-time embedded systems," in *Proc. of 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2012),* 2012.

[2] R. Garibay-Martínez, G. Nelissen, L. L. Ferreira and L. M. Pinho, "On the Scheduling of Fork-Join Parallel/Distributed Real-Time Tasks," in *Proc. of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES'14), to appear*, 2014.

[3] N. Fisher, S. Baruah and T. P. Baker, "The partitioned scheduling of sporadic tasks according to static-priorities," in *Proc. of the IEEE 18th Euromicro Conference on Real-Time Systems (ECRTS'06),* 2006.

[4] P. Axer, S. Quinton, M. Neukirchner and R. Ernst, "Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints," in *Proc. IEEE 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*, 2013.

[5] M. Qamhieh, F. Fauberteau and S. Midonnet, "Performance Analysis for Segment Stretch Transformation of Parallel Real-time Tasks," in *Proceedings of the 5th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2011)*, 2011.

[6] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and Microprogramming,* vol. 40, no. 2-3, pp. 117 - 134 , 1994.

Towards Non-invasive Run-time Verification of Real-Time Systems

Ricardo C. Pinto
LaSIGE/Faculty of Sciences
University of Lisbon
ricardo.pinto@fc.ul.pt

José Rufino
LaSIGE/Faculty of Sciences
University of Lisbon
ruf@di.fc.ul.pt

***Abstract*—Support for Run-time Verification (RV) has mostly been provided by software mechanisms, via the instrumentation of code for observing (monitor) and handling deviations from specification. Although this approach is fitting for some domains, it can have a nefarious influence in embedded real-time systems, impacting the system from the analysis to the operation stages.**

A novel alternative to code instrumentation is the embedding of such mechanisms directly in hardware, thus negating the impact in system properties, namely timeliness. The availability of soft-processors and companion System-on-a-Chip (SoC) Intellectual Property cores enable the hardware-based approach to RV.

This paper addresses the foundations for RV support via hardware mechanisms. A flexible observer entity is defined, to be merged into a SoC architecture. Monitoring is performed at the SoC bus that interconnects processor and peripherals, enabling the gathering of information regarding events of interest occurring during system execution and relaying it to external entities for handling.

I. Introduction

Run-time Verification (RV) is a well-know technique in the software world to perform the verification of a system. It is applied to a software design, to be used throughout its life-cycle stages - from early verification to operational deployment. The cornerstone of RV is the monitoring of values and events, and then comparing them to a given specification.

The classical approach to RV has been through *code instrumentation*. The system software is instrumented with specific functions which are not part of the functional specification of the system. These functions are executed during run-time, monitoring and assessing the state of the system, i.e. its adherence to the functional requirements. Instrumented code is therefore closely intertwined with the code dictating the progression of the system itself (*see* Figure 1).

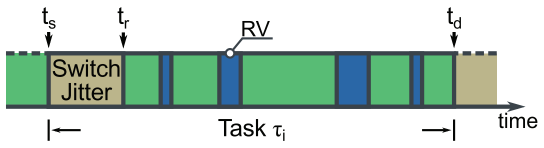

Fig. 1: Task Switching and Execution with RV

This work was partially supported by the EC, through project IST-FP7-STREP-288195 (KARYON) and by FCT, through project PTDC/EEI-SCR/3200/2012 (READAPT), by LaSIGE Strategic Project PEst-OE/EEI/UI0408/2014, and Doctoral Grant SFRH/BD/72005/2010.

The usage of current RV techniques in real-time systems is a double-edged sword: whilst the systems can benefit greatly from RV, the overhead imposed by code instrumentation raises issues stemming in system design up to analysis and operation. At system design, adding code to a real-time task implies a higher Worst-Case Execution Time (WCET) which must be computed for schedulability purposes. At operation, the observation of the system interferes with the system itself due to the *observer effect*[1].

An illustration of such disturbances can be seen in Figure 1, where a piece of code has been added to measure the switching time of a real-time task. The task τ_i has a computed switching time at t_s. The effective time of switching is $t_r = t_p + t_j$, where t_j is the time added by execution jitter, including RV code. There are additional disturbances caused by other RV statements, which result in a higher WCET than a task without RV.

A solution for the RV issues raised by code instrumentation comes from the hardware domain. The growth in the usage of reconfigurable logic supporting System-on-a-Chip (SoC) designs (soft-processors and peripherals) enables the design of innovative solutions to address issues raised by software, and RV is no exception. The inclusion of non-invasive, hardware-based RV mechanisms negates the penalties in timeliness and performance, thus providing results with higher accuracy and without any impact on the system itself.

This paper presents the current work on designing and implementing a hardware-based observer entity aiming at non-intrusive event monitoring. Such entity will provide the support for effective, non-intrusive RV in embedded real-time systems. The remainder of this paper is organized in the following manner: Section II define the System Model to be used in the definition of an observer entity for RV; Section III details the design, specification and implementation in a SoC platform of such observer entity; Section IV presents related work in hardware support for RV and Section V concludes this paper discussing future work directions for achieving robust RV.

[1]The *observer effect* designation stems from physics, where the act of observing a phenomenon interferes with its characteristics.

II. System Model

The observer entity is to be implemented in an embedded real-time system, consisting of an *execution platform* comprised both of hardware and software. Therefore, the definition of such platform is in order. Furthermore, a definition of an *event* is also necessary, formalizing what should be captured by the observer entity.

A. Real-time System Execution

The execution of a real-time system relies on two supporting platforms: *software*, through a Real-Time Operating System (RTOS) providing scheduling and dispatching facilities, together with system primitives for Input/Output (I/O) activities; *hardware*, through an embedded computing platform supporting the execution of the software entity.

1) Hardware: Current embedded systems are implemented resorting to computing platforms which are integrated in a single integrated circuit. An instance of such computing platform is a microcontroller, which has a processing element (CPU) and several I/O peripherals to exchange data with the environment or other systems. Such systems are known as System-on-a-Chip (SoC), due to its level of integration. A diagram showing such a system in presented in Figure 2.

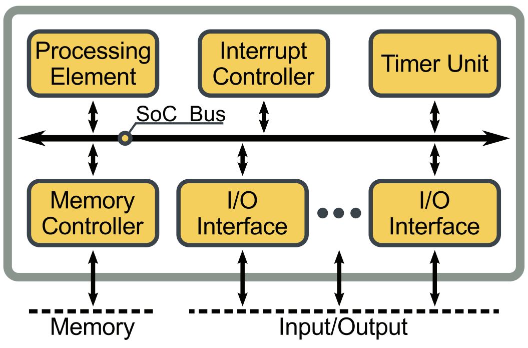

Fig. 2: Generic SoC Computing Platform Architecture

The hardware platform provides the resources necessary for the software entity: *Processing Element*, offering the processor resources; *I/O Interfaces*, exchanging data with external systems and/or the environment; *Memory*, to hold the software executable and state (variables); *Timer Unit*, providing the system with the ability to count time; *Interrupt Controller*, managing the interrupt requests coming from peripherals and feeding them to the processing element.

These components are interconnected through a SoC bus, in a (multi-)master/slave. Components are memory-mapped, with each component on the SoC being accessed through a range of addresses. The operations to be performed are either read or write, in a similar fashion to a memory device. These operations are initiated through master components, which are the only ones allowed to initiate a new transfer. The bus also embeds the interrupt request lines, allowing the slave devices to signal the masters of their need to communicate, e.g. signal they have data available to be transferred to the master.

All instructions to be executed by the processing element have to pass through the bus, coming from the memory component. The transfer of data through the bus can be modelled as $Bus_{trx} \overset{def}{=} (address, data, operation)$, where: *address* is the value of the addressed component; *data* is the value of the data being exchanged; *operation* is the direction of the data, e.g. read or write. Additional control signals, e.g. transfer length, are not of interest for monitoring purposes. Interrupts, Bus_{int}, do not carry additional information.

2) Software: The software platform comprises a set of tasks τ_i, mapping the intended functional specification into software. The execution of the tasks is supported by a RTOS. The usage of a RTOS provides the scheduling and dispatching of the tasks together with primitives to perform I/O activities, inter-task synchronization and communications.

Task execution cannot be decoupled from RTOS execution. Every time a task invokes a system primitive, RTOS facilities are used to fulfill its function, thus deviating the task execution from its designed flow. Furthermore, a scheduling function is performed periodically, deciding which task should be given processor resources. A companion dispatch function performesa the actual task switching. An illustration with an example of these is shown in Figure 3.

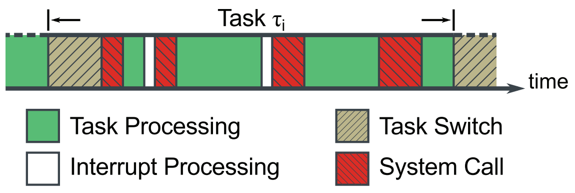

Fig. 3: Generic Task Execution Model

During task execution in an RTOS, there are at least three components: *Task Switch*, when the RTOS dispatches the task to be run; *Task Processing*, with code of the task itself; *System Call* for I/O activities or inter-task synchronization and communication. An additional component is the *Interrupt Processing*, which is executed upon interrupt signalling.

B. Events

An *event* is the information gathered through monitoring of a parameter of interest, e.g. memory address, interrupt line. An event ϵ is defined as a tuple $\epsilon \overset{def}{=} (t, s, i)$ where: t is the time of occurrence; s is the source of the event; i is the specific information pertaining to the event. The time t increases monotonically, establishing event causality. Two or more events may occur at the same time, i.e. $t_{k+1} \geq t_k$.

Broadly speaking, events can have two origins: *hardware*, such as interrupts, memory accesses; *software*, such as values of variables, flow of execution (instructions) or interrupt handling routines.

The identification of events, both in time and source is the basic functionality required for monitoring. Additional information regarding the event enrich the quality of the information provided by the monitoring, and thus lead to the support of RV mechanisms.

III. Observer Entity

The observer does not interfere with the behaviour of the observed system, thus negating the observer effect. The properties of the observer are: **non-intrusive**, not requiring code instrumentation nor affecting system operation; **configurable**, being able to accommodate different event triggers.

A. *Design*

Using the previously defined system model as working basis, an Observer Entity (OE) is defined, to be integrated in an embedded computing platform equivalent to the one presented in Figure 2. The ability to connect to an internal bus architecture is crucial, enabling the observation of data transfers and signalling taking place inside the computing platform, namely instructions and interrupts.

The ability to monitor interrupts and memory addresses allows to measure the latency of a task switch, from the instant the Timer Unit signals the passing of a tick to the scheduler to the time the task switch is completed. The design of an entity with the previous requirements results in the architecture shown in Figure 4.

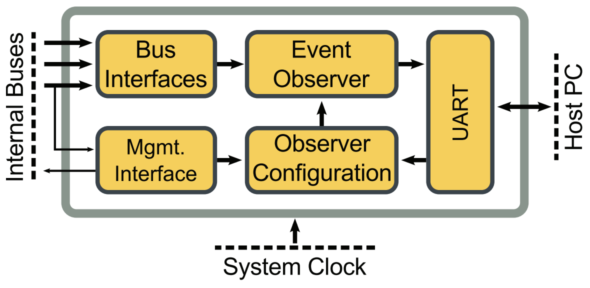

Fig. 4: Observer Entity Architecture

The OE shown in Figure 4 is plugged to the internal buses of a SoC architecture, and is comprised of several modules: *Bus Interfaces*, managing the physical interface to the buses and performing the detection of bus activity, e.g. bus transfer or interrupt; *Management Interface*, handling the support for configuration via the bus itself; *Observer Configuration*, storing the aforementioned configuration, i.e. which events should be detected; *Event Observer*, detecting events of interest based on the configuration and tagging them to be relayed to other systems; *UART*[2], providing an Out-of-band (OOB) interface for relaying the detected events to another system, e.g. a Personal Computer (PC).

The configuration of the OE can be performed through: the system being monitored itself, preferably upon system initialization; OOB, via the UART. The key-point of the OE is that it can be reconfigured after the system is deployed. Such architecture effectively enables non-intrusive hardware monitoring, with the flexibility of being able to accommodate detection of different events.

The operation of the OE is performed at every hardware clock cycle, synchronously with the bus. The OE continuously monitors the bus to detect the start of a new bus transfer operation, or the assertion of an interrupt line. The operation of the OE is described by the algorithm shown in Algorithm 1.

Algorithm 1: Event Monitoring

```
Input: System clock hardware_clock_tick
Output: Event evt
Initialize(Config)
foreach hardware_clock_tick do
    numTicks ← numTicks + 1
    if newEvent(Bus) then
        if $\exists\, id \in Config : Config[id] = Bus_{trx}.address$ then
            evt.time ← numTicks
            evt.source ← Config[ID].source
            evt.info ← $Bus_{trx}.data$
            outputEvent(evt)
        foreach $id \in Config : Config[id] = Bus_{int}$ do
            evt.time ← numTicks
            evt.source ← Config[ID].source
            evt.info ← null
            outputEvent(evt)
```

The tick count `numTicks` increases monotonically with each hardware clock tick. For a 50 MHz clock, it increases every 20 ns. If bus activity is detected, the configuration table is checked and if there is a match, the event is tagged with: timestamp, source and info, e.g. data in a memory access.

B. *Implementation*

The implementation of the OE architecture shown in Figure 4 is being performed in VHDL[3], and integrated in a LEON3 SoC [1], which includes the LEON3 soft-processor. The LEON3 processor implements the SPARC V8 Instruction Set Architecture (ISA) [2], and is connected to the peripherals through ARM Advanced Microcontroller Bus Architecture (AMBA) [3]. A diagram showing the LEON3 SoC together with the Observer is shown in Figure 5.

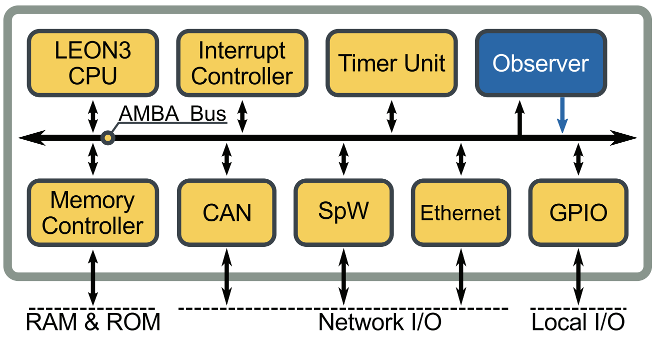

Fig. 5: Observer Entity on a LEON3 System-on-a-Chip

The SoC design provides several IP cores implementing I/O interfaces, together with a Memory Controller for Random-Access Memory (RAM) and Read-Only Memory (ROM).

[2]UART - Universal Asynchronous Receiver/Transmitter

[3]VHDL stands for Very High Speed Integrated Circuit Hardware Description Language

The AMBA bus is a high performance multi-master bus. Data is transferred in parallel, with one transfer per clock-cycle. Addressing is performed through memory-mapping, where each component in the SoC is seen as a range of memory addresses. The memory where software code resides is no exception, exposing its accesses to hardware monitoring.

The specification [3] defines two types of bus: AMBA High-performance Bus (AHB), for high-throughput components, e.g. CPU, RAM, Ethernet; Advanced Peripheral Bus (APB), for low throughput components, e.g. UART, General Purpose I/O (GPIO). Both these buses are connected to the OE, enabling monitoring of data exchange between SoC components.

The implementation present in the LEON3 SoC embeds the interrupt request lines of the peripherals in the AMBA bus. Such embedding eases the monitoring of both data transfers and interrupt requests, since both are available through the same bus interface.

The logical structure of the information pertaining to a monitored event is shown in Figure 6, described in VHDL.

```
-- Event
type event_t is record
  Time       : timestamp_t;     -- Time of occurrence
  Source     : event_source_t;  -- Source
  Info       : event_info_t;    -- Specific Data
end record event_t;

-- Event Source
type event_source_t is record
  ID     : integer; -- ID: Task t1, Interrupt 15
  Class  : integer; -- Classes, e.g. hw or sw
end record event_source_t;

-- Event Specific Data
type event_info_t is record
  address    : address_t;    -- Address
  data       : data_t;       -- Data
  operation  : operation_t;  -- Operation, e.g. read
end record event_info_t;
```

Fig. 6: VHDL Description of Event Information

The `timestamp_t` data type should be wide enough to avoid rollover of time information, and is dependent on the system clock frequency. The rollover can be seen as a violation of the monotonicity, where time goes backward. The `Source` field stores data pertaining to the source of the event. Furthermore, it can be configured with the `Class` extra field, to provide context for the event specific data field, `Info`. The event contained in the format shown in Figure 6 is converted into a stream of bits to be transmitted by the UART via a VHDL function.

C. Event Data Output & Exploitation

The usage of an UART as an OOB mechanism enables the processing and exploitation of the generated event data on an external system, such as a PC. Therefore, the OE can be used to: characterize the system during the Verification & Validation (VV) stage, including timeliness; log the behavior for performance and timeliness assessment; enable proactive fault-tolerance mechanism design and execution.

The data output format is configurable via the Observer Configuration module, and can be formatted to be directly used by specific verification and visualization tools. The formatting is performed directly in hardware, by using specific serializing functions to convert the storage representation of Figure 6 into a stream of bits to be output by the UART. Output of raw data together with software filtering on the host system is also possible, enabling flexible monitoring data exploitation.

An envisaged visualization tool to be used with the OE is Grasp [4]. Grasp was designed for the visualization of real-time system execution, namely task execution and switching. Such a tool enables the verification of scheduling information, by visualizing the task execution and preemption points.

IV. Related Work

The design of hardware RV mechanisms has been receiving a growing interest. A first approach to monitoring was introduced in [5], with minimal code instrumentation. The approach in [6] solved the instrumentation issue, using a dedicated CPU.

Some of the the approaches address both the issue of monitoring and verification in a single instance [7]. The verification procedure is mapped into soft-microcontroller units, embedded within the design, and use formal languages such as past-time Linear Temporal Logic (ptLTL) for verification, with clauses checked by a CPU embedded in the design.

V. Conclusion and Future Work

Online monitoring and Run-time Verification (RV) of embedded real-time systems is a topic which is expected to grow in the coming years, thrusted by the design of autonomous control applications. The application of RV to real-time systems, however, brings an overhead which may be too costly, due to the impact in timeliness. The availability of soft-processors and SoC designs opens room for novel monitoring and RV, supporting non-invasive hardware-based RV for autonomous applications.

The provision of a reconfigurable and non-invasive Observer Entity (OE) for monitoring is the first step towards more sophisticated mechanisms and services. The data gathered by the OE can be used to feed verification clauses, thus enabling flexible non-invasive hardware-based RV.

References

[1] *GRLIB IP Library Users Manual*, Aeroflex Gaisler A.B., Apr. 2014. [Online]. Available: http://gaisler.com/products/grlib/grlib.pdf

[2] *The SPARC Architecture Manual*, SPARC International Inc., 1992.

[3] ARM Limited, *AMBA Specification*, ARM Specification 2.0, May 1999.

[4] M. Holenderski, M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 37–42.

[5] M. El Shobaki and L. Lindh, "A Hardware and Software Monitor for High-level System-on-Chip Verification," in *2001 International Symposium on Quality Electronic Design*, 2001, pp. 56–61.

[6] J. C. Lee, A. S. Gardner, and R. Lysecky, "Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems," *Engineering of Computer-Based Systems, IEEE International Conference on the*, vol. 0, pp. 52–60, 2011.

[7] T. Reinbacher, M. Függer, and J. Brauer, "Runtime verification of embedded real-time systems," *Formal Methods in System Design*, pp. 1–37, 2013.

Notes