CRTS 2014

Proceedings of the 7^{th} International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems

Rome, Italy
December 2, 2014

In conjunction with:
The 35^{th} International Conference on Real-Time Systems (RTSS'14),
December 3-5, 2014

Edited by Reinder J. Bril and Jinkyu Lee

Foreword

Welcome to Rome and the 7^{th} International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2014). The CRTS workshops provide a forum for researchers and technologists to discuss the state-of-the-art, present their work and contributions, and set future directions in compositional technology for real-time embedded systems.

CRTS 2014 is organized around presentations of papers (regular papers and invited extended abstracts) and a panel discussion focussed on the "state-of-the art and future directions" of CRTS. As usual, the presentations of regular papers address typical topics of CRTS. The invited presentations, on the other hand, particularly aim at open problems ("future directions") and give an indication about the difficulty to solve these problems. These latter presentations may be controversial or thought provoking, but also be an invitation to join in tackling hard problems. In addition, they are meant to serve the organizing committee with respect to future directions for CRTS.

A total of 7 papers were selected for presentation at the workshop, 2 regular papers and 5 invited extended abstracts. These proceedings are also published as a Computer Science Report from the Technical University of Eindhoven (CSR-1407) available at http://library.tue.nl/catalog/CSRPublication.csp?Action=GetByYear.

This year, CRTS is organized in conjunction with the 5^{th} Analytical Virtual Integration of Cyber-Physical Systems Workshop (AVICPS 2014), which has close theoretical and practical scientific interests. Our joint program contains a keynote by Prof. Dr. Dr. h.c. Manfred Broy from the Technisch Universität München.

We would like to thank the Organizational Committee listed below, for granting us the honor, privilege and opportunity to be the co-chairs of CRTS 2014.

Insup Lee	University of Pennsylvania, USA
Thomas Nolte	Mälardalen University, Sweden
Insik Shin	KAIST, Republic of Korea
Oleg Sokolsky	University of Pennsylvania, USA

Moreover, we would like to thank the Technical Program Committee listed below, for their work in reviewing the regular papers and extended abstracts, and helping to make the workshop a success.

Benny Åkesson	Czech Technical University in Prague, Czech Republic
Luís Almeida	Universidade do Porto, Portugal
Björn Andersson	Software Engineering Institute at Carnegie Mellon University, USA
Moris Behnam	Mälardalen University, Sweden
Enrico Bini	Scuola Superiore Sant'Anna, Italy
Arvind Easwaran	Nanyang Technological University, Singapore
Martijn M.H.P. van den Heuvel	Eindhoven University of Technology (TU/e), The Netherlands
Hyun-Wook Jin	Konkuk University, Republic of Korea
Julio Luis Medina Pasaje	Universidad de Cantabria, Spain
Jan Reineke	Saarland University, Germany
Luca Santinelli	ONERA, France
Mikael Sjödin	Mälardalen University, Sweden
Linh Thi Xuan Phan	University of Pennsylvania, USA
Lothar Thiele	Swiss Federal Institute of Technology Zurich, Switzerland
Tullio Vardanega	Università di Padova, Italy

Last but not least, special thanks go to the RTSS 2014 Workshop Chair, Program Chair and General Chair listed below, as well as the AVICPS 2014 co-chairs, for their support and assistance in organizing this joint seminar.

Rodolfo Pellizzoni	University of Waterloo, Canada	(RTSS 2014 Workshops Chair)
Christopher D. Gill	Washington University in St. Louis, USA	(RTSS 2014 Program Chair)
Michael González Harbour	Universidad de Cantabria, Spain	(RTSS 2014 General Chair)
Sibin Mohan	University of Illinois at Urbana-Champaign	(AVICPS 2014 Co-chair)
Jean-Pierre Talpin	INRIA, France	(AVICPS 2014 Co-chair)

Jinkyu Lee and Reinder J. Bril
Co-chairs
7^{th} International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2014)

Table of Contents

Regular papers

Extended Abstracts

Supporting Fault-Tolerance in a Compositional Real-Time Scheduling Framework

Guy Martin Tchamgoue[1], Junho Seo[1], Jongsoo Hyun[2], Kyong Hoon Kim[1], and Yong-Kee Jun[1]

[1]Department of Informatics
Gyeongsang National University
660–701, Jinju, South Korea

[2]Avionics SW Team
Korea Aerospace Industries, Ltd.
Sacheon, South Korea

guymt@ymail.com, joy2net@gnu.ac.kr, ksjh0111@koreaaero.com, {khkim,jun}@gnu.ac.kr

ABSTRACT

Component-based analysis allows a robust time and space decomposition of a complex real-time system into components, which are then recomposed and hierarchically scheduled under potentially different scheduling policies. This mechanism is of great benefit to many critical systems as it enables fault isolation. To provide fault-tolerant scheduling in a compositional real-time scheduling framework, a few works have recently emerged, but remain inefficient in providing fault isolation or in terms of resource utilization. In this paper, we introduced a new interface model that takes into account the fault requirements of a component, and a fault-tolerant resource model that helps the component to effectively respond to each of its child components in presence of a fault. Finally, we analyzed the schedulability of the framework considering the Rate Monotonic scheduling algorithm.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: Fault tolerance; D.4 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Theory, Reliability

Keywords

Compositional real-time scheduling, periodic resource model, periodic task model, fault-tolerant scheduling

1. INTRODUCTION

The increasing size and complexity and the requirement of high performance have led to the rapid adoption of the component-based analysis in many cyber-physical systems. A compositional real-time scheduling framework allows multiple components, that may have been individually developed and validated, to be hierarchically composed and scheduled altogether. In this kind of open computing environment, a component or partition receives computational resources from its parent component and shares the resources with its child components through its own local scheduler. This robust space and time partitioning opens ways to achieving rigorous fault containment. Therefore, faults can transparently be detected and handled by a fault management policy at each level of the hierarchy: *intra-component* (or *task level*), *inter-component* (or *component level*), and *system level* .

In safety-critical real-time systems, such as avionics [1] and automotive [2], where component-based analysis has become a standard, two main conflicting challenges are to be addressed: (1) providing an efficient resource sharing for economical reasons, and (2) guaranteeing the reliability of the system for validation and certification. Many compositional real-time scheduling frameworks [3, 5, 15, 16, 17] have already been proposed, but with a great focus on efficient resource abstraction and sharing, schedulability analysis, and abstraction and runtime overheads. Thus, research on a fault-tolerant compositional real-time scheduling framework is yet to be done. Such a framework should provide an efficient resource model for an effective resource sharing even in presence of faults. Nevertheless, many error recovery strategies such as redundancy [7, 11, 14], roll-back [6, 13, 19] and roll-forward [12, 18] with check-pointing [4], have already been devised for the long studied field of fault-tolerance in real-time systems, but their direct application to a compositional scheduling framework has not been thoroughly investigated.

Considering the periodic resource model [16], Hyun and Kim [8] proposed a task level fault-tolerant framework and later extended it with a component level fault containment with backup partitions [9]. Although it offers task and component level fault isolation, the approach remains inefficient as the highest possible resource is always required to guarantee the feasibility of the system even in the absence of faults. Jin [10] extended the periodic resource model to support the backup resource requirements, but does not provide a fine-grained fault management as the system definitely switches to a backup partition whenever a fault is detected inside the associated primary partition.

In this paper, we propose a new compositional real-time scheduling framework that uses the time redundancy technique to tolerate faults. Our framework introduces a new interface model that takes into account the real-time fault requirements of a component, and a resource model that helps the component to effectively respond to each of its child components in presence of a fault. When a fault is detected inside a component, the new resource model guarantees to provide an extra resource to the faulty component

only until the fault is handled and thereafter, switches back to a normal supply as the demand of the component has also decreased. Contrarily to the previous approaches [8, 9, 10], the new model provides a more flexible and efficient resource sharing in presence of faults. The schedulability of the framework has been analyzed considering the Rate Monotonic (RM) scheduling algorithm. However, our analysis focuses only on errors that are caused by transient faults, allowing each single task of the system to define its own error recovery strategy.

The remainder of this paper is organized as follows. Section 2 presents our system model with an overview of a compositional real-time scheduling framework, and describes the problems addressed in the paper. Section 3 focuses on the proposed framework itself and introduces the new interface and resource models, and discusses the schedulability analysis with the RM algorithm. Section 4 provides details on how to compute each parameter that makes up the fault-tolerant interface model. Finally, the paper is concluded in with Section 5.

2. BACKGROUND

This section presents our system model with an overview of a compositional real-time scheduling framework (CRTS), describes our fault model and finally defines the problems handled in this paper.

2.1 System Model

In a compositional real-time scheduling framework [16, 17], components are organized in a tree-like hierarchy where a upper-layer component allocates resources to its child components, as shown in Figure 1. Thus, the basic scheduling unit (i.e. component or partition) of the framework is defined as $C(W, R, A)$, where W is the workload, R the resource model supported by the upper-layer component, and A the scheduling algorithm of the component.

In this paper, we assume that the workload W of each component is composed of a set of periodic real-time tasks running on a single processor platform. Each task τ_i is then defined by its period, p_i and its worst-case execution time, e_i. We also assume the deadline of each task τ_i to be equal to its period p_i.

A *resource model* R specifies the exact amount of resource to be allocated by a parent component to its child components. The periodic resource model $\Gamma(\Pi, \Theta)$ [16], as in Figure 1, guarantees a resource supply of Θ at every period of Π time units to a given component. In contrast to the resource model, the *interface model* abstracts a component together with its collective real-time requirement as a new real-time task. The periodic interface model $I(P, E)$ [16] represents a component task I with execution time E and period P.

As an example, Figure 1 depicts a two-layer compositional real-time scheduling framework comprising three components, C_0, C_1, and C_2. The two tasks of component C_1 which are scheduled with EDF (Earliest Deadline First) are abstracted under the interface I_1 as a periodic task with a period of 10 and an execution time of 3 time units. Similarly, component C_2 which contains two tasks scheduled with RM (Rate Monotonic) is seen by the upper layer component as a single task represented by I_2. Thus, component C_0, which is also summarized as interface I_0, focuses on scheduling C_1 and C_2 as simple real-time tasks through their respective

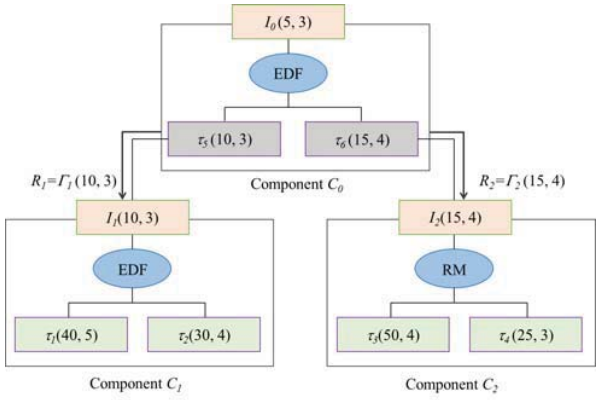


Figure 1: An example of compositional real-time scheduling framework

interfaces I_1 and I_2, therefore providing C_1 and C_2 with resource models R_1 and R_2, respectively.

2.2 Fault Model

In this paper, we consider only errors that are caused by transient faults. We assume that only the single task under execution at the time of a fault occurrence is affected by the fault. Whenever a fault is detected, the state of the affected task is recovered by an appropriate error recovery strategy such as redundancy, rollback, or roll-forward. Therefore, we expect each task τ_i to define its own recovery strategy and thus, maintains its own backup task referred to as β_i. For any task τ_i, the backup execution time, denoted by b_i, is assumed to be not greater than the normal execution time e_i (i.e. $0 \leq b_i \leq e_i$). The backup execution is defined according to the recovery strategy as follows:

- $b_i = e_i$: when the re-execution strategy is applied,
- $0 < b_i < e_i$: for a forward recovery strategy such as an exception handler
- $b_i = 0$: when the fault is to be ignored.

A fault is assumed to be detected at the end of the execution of each task as this represents the worst-case scenario. Once a fault is detected on a task τ_i, its backup task β_i is to be released and executed by the task's deadline. Thus, a task τ_i is supposed to finish at least by $(p_i - b_i)$ in order to make enough slack time for its backup task. However, due to the nature of the resource model, the remaining slack time of b_i may still be insufficient to cover the backup task, in which case we assume the recovery to start from the next period of the task. With a periodic resource model $\Gamma(\Pi, \Theta)$ for example, the system may become non schedulable because the resource supply of $\left\lfloor \frac{b_i}{\Pi} \right\rfloor \Theta$ cannot satisfy the backup requirement of b_i time units for a faulty task τ_i. We also assume a fault to occur only once in a time interval of T_F units, which represents the minimum distance between two consecutive faults in the system. When a fault is detected on a task τ_i, the faulty component may require an extra computational resource to cover up the fault. However, due to the periodicity of the resource supply and in order to preserve the schedulabilty of other components in the framework, the

extra resource can only be claimed from the next resource period. Thus, each component of the framework is assumed to detect a task fault only at the end of each resource supply. Therefore, the component assumes the fault recovery process to start from the resource period that comes right after the one in which the fault was detected. It is important to emphasize on the fact that the backup task does not need to wait until the next resource period to be executed, but as soon as it gets ready.

2.3 Problem Statement

In this paper, we present a fault-tolerant compositional scheduling framework assuming the periodic real-time task model. Considering a single fault model, we propose a task level fault management scheme while handling the following problems:

- *Interface model:* to model the workload W of a component $C(W, R, A)$ as a single periodic task with consideration of the deadline and fault requirements of each task. An upper-layer component can then use the interface model to efficiently share its resource with its child components.
- *Resource model:* to guarantee an optimal resource supply to each component in order to satisfy its deadline and fault-tolerance requirements.
- *Interface generation:* to effectively determine each parameter that makes up the interface model for each component of the framework.
- *Schedulability analysis:* to guarantee to each component especially in the presence of faults, the minimum resource supply that makes it schedulable.

We believe that such a fault-tolerant system will be useful for example in the design of a modern avionics mission computer that implements a strict time and space partitioning based on the ARINC 653 standard [1]. In such a system faults need to be handled and dealt with properly. A single fault may, for example, cause an entire operational flight program to behave incorrectly or to fail, eventually forcing the mission computer itself to a cold or warm restart. A warm restart of the system takes about 5 seconds, which may then force ongoing missions such as target attack and aerial reconnaissance to abortion [8].

3. FAULT-TOLERANT CRTS

This section describes a new fault-tolerant compositional real-time scheduling framework. We present our new interface and resource models. The schedulability analysis of the framework is provided assuming the Rate Monotonic (`RM`) scheduling algorithm.

3.1 Interface Model

Each component of the framework contains a *Fault Manager* (FM) module which function is to detect and handle faults inside the component. Although this paper considers only the `RM` algorithm, any other scheduler capable of handling faults like `EDF` maybe used. A new periodic interface model defined by $I(P, E, B, M)$ is introduced to support both the real-time and the fault requirements of each component. In this interface definition, P, E, and B respectively

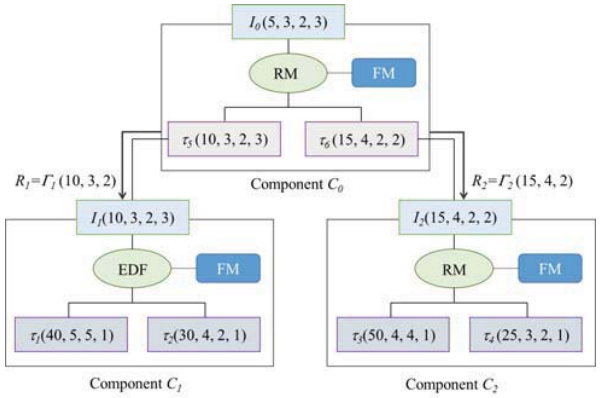


Figure 2: The proposed scheduling framework

represents the period, the execution time during the normal mode, and the extra execution time to be supplied in case of fault for backup. When a fault is detected on a task, the component may require more than one resource supply to recover from the fault. Therefore, the parameter M is to materialize the total number of resource intervals which are needed by the component to properly respond to faults. Thus, when a fault signaled inside a component $C(W, R, A)$, the overall resource demand of the component, due to the release of a backup task, increases by approximately $M \times B$ time units. In other words, when a fault is detected on a task τ_i inside a component $C(W, R, A)$, the component level backup task $I_b(P, B)$ will be released M times in order to request enough resource to cover up the backup requirement of the faulty task τ_i. We also normalized the definition of a task τ_i to add a new parameter m_i which as M, represents the number of backup releases of the task. If a fault occurs on a task $\tau_i(p_i, e_i, b_i, m_i)$, the additional backup job with b_i execution times is released for exactly m_i times. With this definition, the backup task β_i of a task τ_i can be registered to spread across multiple release periods.

An example of the new framework is shown in Figure 2, where component C_2 has two periodic tasks $\tau_3(50, 4, 4, 1)$ and $\tau_4(25, 3, 2, 1)$ scheduled with a fault-tolerant `RM` algorithm. The component exposes its interface $I_2(15, 4, 2, 2)$ to its parent component C_0 to claim a computational time of 4 units every 15 time units under normal execution. However, if a fault is detected, C_2 will require an additional 2 time units to be supplied during the next 2 resource periods in order to deal with the fault. In a similar way, component C_1 presents its interface $I_1(10, 3, 2, 3)$ to C_0, which then focuses on scheduling the two components as two normal periodic tasks.

3.2 Resource Model

This paper introduces a new fault-aware periodic resource model which extends the existing periodic resource model [16] to support faults in a compositional scheduling framework. The fault-tolerant resource model $\Gamma(\Pi, \Theta, \Delta)$ guarantees to supply to each component a resource amount of Θ time units whenever the component is running without any fault. However, when a fault is detected on a task τ_i, the resource demand of the component increases by b_i. To support the fault recovery process, the component is supplied an addi-

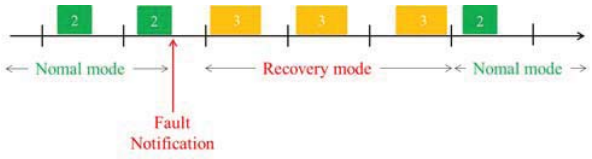


Figure 3: An example of resource supply for $\Gamma(5,2,1)$ where $M=3$

tional computational time of Δ. Thus, the fault-tolerant resource model $\Gamma(\Pi,\Theta,\Delta)$ supplies Θ time units during the normal execution and increases the supply to $\Theta+\Delta$ during the recovery time. Contrarily to the previous fault-tolerant model $\Gamma(\Pi,\Theta^p,\Theta^b)$ [10] that provides Θ^p during the normal execution and definitely switches to Θ^b when a fault is detected, our resource model $\Gamma(\Pi,\Theta,\Delta)$ switches back to the normal execution mode when the fault is entirely recovered and therefore, continues to supply only Θ time units. The exact number of time the extra resource is supplied is just taken from the interface of each component. Figure 3 shows an example of resource supply model $R=\Gamma(5,2,1)$ to a component $C(W,R,A)$ with the interface model $I(5,2,1,3)$.

For the schedulability purpose, it is important to accurately evaluate the amount of resource supplied by a resource model to a component. The supply bound function $\mathtt{sbf}_\Gamma(t)$ of a resource model Γ calculates the minimum resource supply for any given time interval of length t. In a normal execution mode, the supply bound function is similar that of the periodic resource model [16] and given by the following equation:

$$\mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t)=\begin{cases} t-(k+1)(\Pi-\Theta) & \text{if } t\in[(k+1)\Pi-2\Theta,\\ & \quad (k+1)\Pi-\Theta],\\ (k-1)\Theta & \text{otherwise,}\end{cases} \tag{1}$$

$$\text{where } k=\max(\lceil(t-(\Pi-\Theta))/\Pi\rceil,1).$$

However, during the recovery mode, the resource supply to the faulty component increases by Δ time units. Thus, the supply bound function for the recovery mode $\mathtt{sbf}^R_\Gamma(t)$ is given by Equation (2).

$$\mathtt{sbf}^R_{\Gamma(\Pi,\Theta,\Delta)}(t)=\mathtt{sbf}_{\Gamma(\Pi,\Theta+\Delta)}(t-\Delta) \tag{2}$$

Given a component $C(W,R,A)$ represented by its interface $I(P,E,B,M)$ and a resource supply model $R=\Gamma(\Pi,\Theta,\Delta)$, if we assume that a fault is detected during the k-th resource supply to C, then the supply bound function can be provided by Equation (3).

$$\mathtt{sbf}_{\Gamma(\Pi,\Theta,\Delta)}(t,k)=\begin{cases}\mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t) & \text{if } t\le t_N\\ \mathtt{sbf}^R_{\Gamma(\Pi,\Theta,\Delta)}(t)-h_s & \text{if } t_N<t\le t_R\\ \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t)+v_s & \text{Otherwise}\end{cases} \tag{3}$$

$$\begin{aligned}\text{where}\quad & t_N=k\Pi-\Theta\\ & t_R=(M+k)\Pi-\Theta\\ & h_s=\mathtt{sbf}^R_{\Gamma(\Pi,\Theta,\Delta)}(t_N)-\mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_N)\\ & v_s=\mathtt{sbf}^R_{\Gamma(\Pi,\Theta,\Delta)}(t_R)-\mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_R)-h_s\end{aligned}$$

EXAMPLE 3.1. *Let us consider a component $C(W,R,A)$ where $W=\{\tau_1(10,1,1,1),\tau_2(15,2,2,1)\}$ and $A=$ RM. Let the interface of the component be given by $I(5,2,1,2)$ and its resource supply modeled by $R=\Gamma(5,2,1)$. Figure 4 compares $\mathtt{sbf}_{\Gamma(\Pi,\Theta,\Delta)}(t,k)$ for $k=1,2,3$, and 4 with the worst-case resource supply of $\Gamma(5,3)$ as considered by the previous work [8, 10]. The new resource model provides a significant gain in terms of resource for the framework. Figure 4 shows that the curves of our fault-tolerant resource model are always between those of the normal and the worst-case resource supply.*

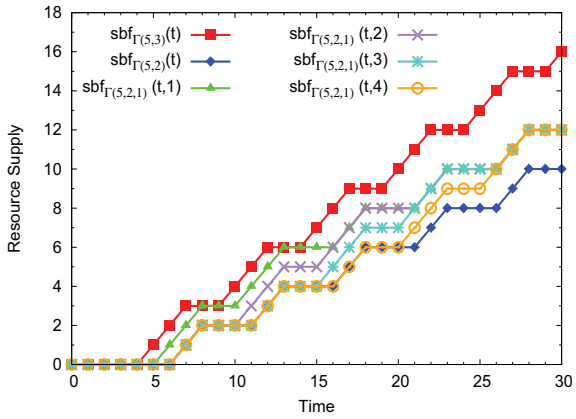


Figure 4: Supply bound function for $\Gamma(5,2,1)$ with $M=2$

3.3 Schedulability Analysis

For the analysis of the schedulability, we focus only on the Rate Monotonic (RM) algorithm which assigns higher priorities to tasks with the shortest periods. Thus, without loss of generality, we assume that tasks are sorted in each component in an ascendant order of their periods, that is $p_i\le p_{i+1}$. Also, when released, a backup task β_i inherits the priority of its faulty task τ_i. We define the *resource demand* of a workload as the amount of resource requested by a component to its parent component. The demand bound function $\mathtt{dbf}_W(A,t)$ computes the maximum resource demand required by the workload W when scheduled with the algorithm A during a time interval t. Since we focus only on the RM algorithm, we will omit the scheduling algorithm from the future notations of the demand bound function.

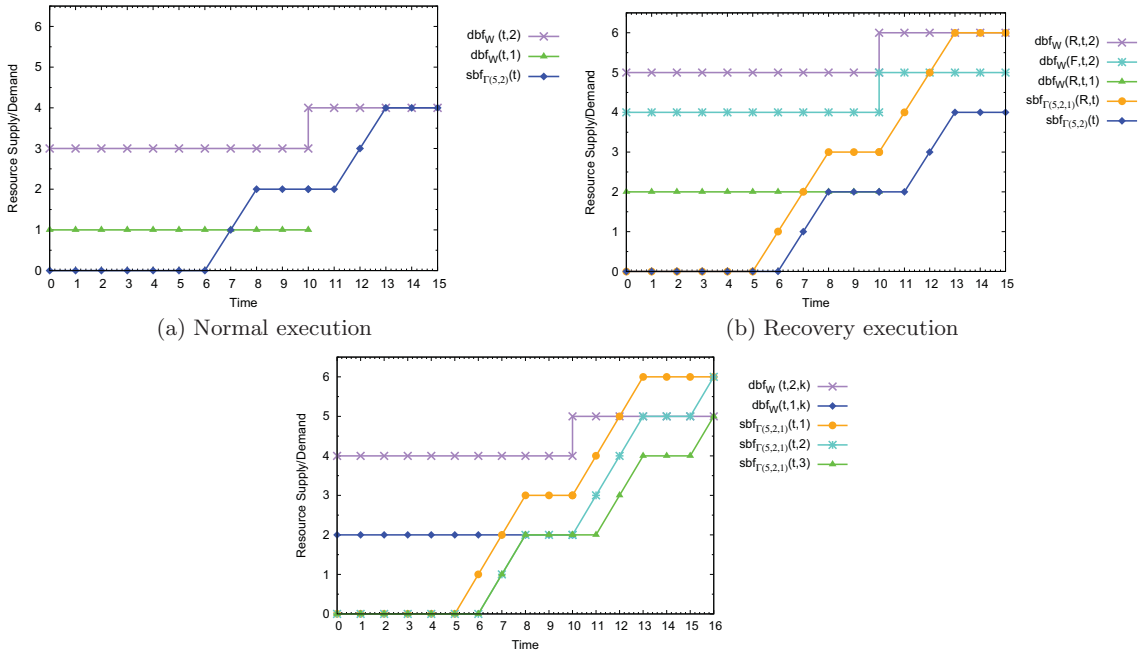
For a component $C(W,R,\mathtt{RM})$ under normal execution, the demand bound function $\mathtt{dbf}_W(t,i)$ of a task τ_i is given by the following equation:

$$\mathtt{dbf}_W(t,i)=e_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{t}{p_j}\right\rceil\cdot e_j \tag{4}$$

where $hp(i)$ represents the set of tasks with priority higher than the one of τ_i. However, if a task τ_i is still recovering from a fault, its demand bound function considering it backup task β_i is given by Equation (5).

$$\mathtt{dbf}^R_W(t,i)=e_i+b_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{t}{p_j}\right\rceil\cdot e_j \tag{5}$$

If the fault is detected on a task τ_j with higher priority than another task τ_i, then the demand bound function $\mathtt{dbf}^F_W(t,i)$



(a) Normal execution

(b) Recovery execution

(c) Fault analysis

Figure 5: Schedulability analysis of Example 3.2

of τ_i is provided by Equation (6). Among all tasks with priority higher than that of τ_i, the demand bound function of τ_i assumes the worst-case situation in which the faulty task τ_j is the one with the maximum backup execution time.

$$\mathtt{dbf}^F_W(t,i) = e_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{t}{p_j} \right\rceil \cdot e_j + \max_{\tau_j \in hp(i)} \left(\min \left(\left\lceil \frac{t}{p_j} \right\rceil, m_j \right) \cdot b_j \right) \quad (6)$$

We can now determine for a task τ_i the demand bound function $\mathtt{dbf}_W(t,i,k)$ assuming that a fault was detected on another task τ_j with priority higher than that of τ_i during the k-th resource supply to the component as in Equation (7).

$$\mathtt{dbf}_W(t,i,k) = e_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{t}{p_j} \right\rceil \cdot e_j + \max_{\tau_j \in hp(i)} \left(\min \left(\max \left(\left\lceil \frac{t-(k-1)\Pi}{p_j} \right\rceil, 0 \right), m_j \right) \cdot b_j \right) \quad (7)$$

A component $C(W,R,\mathtt{RM})$ is schedulable if the resource demand of its workload W is guaranteed to be satisfied by the resource model $R = \Gamma(\Pi,\Theta,\Delta)$ during the normal execution mode and also in presence of a fault as summarized in Theorem 1.

THEOREM 1. *A given component $C(W,R,\mathtt{RM})$ where $W = \{\tau_i(p_i,e_i,b_i,m_i) | i = 1,\ldots,n\}$ and which interface is defined as $I(P,E,B,M)$, is schedulable with a resource model $R = \Gamma(\Pi,\Theta,\Delta)$ if and only if for all $\tau_i \in W$, there exists $t_i \in [0,p_i]$ such that the following three conditions are satisfied:*

1. $\mathtt{dbf}_W(t_i,i) \leq \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_i)$
2. $\mathtt{dbf}^R_W(t_i,i) \leq \mathtt{sbf}^R_{\Gamma(\Pi,\Theta,\Delta)}(t_i)$
3. $\mathtt{dbf}_W(t_i,i,k) \leq \mathtt{sbf}_{\Gamma(\Pi,\Theta,\Delta)}(t_i,k), \forall k = 1,\ldots,(\lceil \frac{p_i}{\Pi} \rceil - 1)$

PROOF. The proof to the first condition of Theorem 1 follows from the work by Shin and Lee [16, Theorem 4.2]. A task τ_i completes its execution requirement at time $t_i \in [0,p_i]$, if, and only if, all the execution requirements from all the jobs of higher-priority tasks than τ_i and e_i , the execution requirement of τ_i , are completed at time t_i. The total of such requirements is given by $\mathtt{dbf}_W(t_i,i)$, and they are completed at t_i if, and only if, $\mathtt{dbf}_W(t_i,i) = \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_i)$ and $\mathtt{dbf}_W(t'_i,i) > \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t'_i)$ for $0 \leq t'_i < t_i$. It follows that a necessary and sufficient condition for τ_i to meet its deadline is the existence of a $t_i \in [0,p_i]$ such that $\mathtt{dbf}_W(t_i,i) = \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_i)$. The entire task set is schedulable if, and only if, each of the tasks is schedulable, which implies that there exists a $t_i \in [0,p_i]$ such that $\mathtt{dbf}_W(t_i,i) = \mathtt{sbf}_{\Gamma(\Pi,\Theta)}(t_i)$ for each task $\tau_i \in W$.

Similarly, the proofs to the two other conditions can be established by repeating the same reasoning with the appropriate demand and supply bound functions. □

EXAMPLE 3.2. *Let us consider again our previous component $C(W,R,\mathtt{RM})$ where $W = \{\tau_1(10,1,1,1),\tau_2(15,2,2,1)\}$ and $R = \Gamma(5,2,1)$. The interface of the component is given by $I(5,2,1,2)$. Figure 5(a) which plots the demand bound function as presented in Equation (4), shows that the component is schedulable for the minimum resource supply of $R = \Gamma(5,2)$. This satisfies the first condition of Theorem 1.*

However, as seen in Figure 5(b), if the resource supply remains $R = \Gamma(5, 2)$ while a fault occurs on task τ_1, task τ_2 will miss its deadlines due to the interference from the backup task of τ_1. Also, if the fault is instead detected on τ_2, the component will still be unschedulable with a resource supply of $R = \Gamma(5, 2)$. However, Figure 5(b) shows that the component becomes schedulable if the resource supply becomes $R = \Gamma(5, 2, 1)$. Figure 5(c) plots the third condition of Theorem 1 to analyze the impact of a faulty τ_1 on task τ_2. It results that by supplying an extra 1 computation time unit during exactly 2 resource period, the component is always schedulable. Therefore, there is no need to always provide C with a resource of $R = \Gamma(5, 3)$. Moreover, Figure 5(c) shows that if the fault is detected during the last resource supply before the deadline of τ_2 (i.e. $[10-15]$), the recovery process will be handled only after the deadline.

4. INTERFACE GENERATION

A component expresses its resource demand to its parent component through its interface which abstracts, without revealing it, the internal real-time requirements of the component. The interface of a component $C(W, R, A)$ is defined by $I(P, E, B, M)$. When a fault is detected on a task τ_i by the local fault manager, the backup task β_i is released to execute for b_i time units. This release also triggers the release of the component backup task as the component is now seen as faulty by its parent component. As a result, the faulty component is provided with an extra Δ time units. The component remains in this faulty status for exactly M resource periods. This section focuses on determining the interface parameters that make each component schedulable with a resource model $\Gamma(\Pi, \Theta, \Delta)$.

In this paper, we assume that the period P of the interface is decided by the system designer. However, there is a tradeoff in choosing the right P for a given component $C(W, R, A)$ due to the scheduling overhead. A smaller P increases the scheduling overhead in the upper-layer component due to the increased number of context switching. Inversely, a larger P makes it also difficult to find a feasible interface model. Thus, we suggest to select P as the minimum period among all tasks in W, or as a number dividing the minimum period, or finally as a common divider to all $p_i, \forall \tau_i \in W$.

The parameter E can be easily determined assuming the component is in its normal execution mode where backup resource supply is not required as stated by the first condition of Theorem 1. When a resource model $\Gamma(\Pi, \Theta, \Delta)$ is provided to a component with interface $I(P, E, B, M)$, the execution time E can be determined by Proposition 1.

PROPOSITION 1. *The schedulability of a given component $C(W, R, \texttt{RM})$ abstracted by the interface $I(P, E, B, M)$, where $W = \{\tau_i(p_i, e_i, b_i, m_i) | i = 1, \ldots, n\}$ and $R = \Gamma(\Pi, \Theta, \Delta)$, is guaranteed if*

$$E = \frac{P \cdot U_N}{\log\left(\frac{2k+2(1-U_N)}{k+2(1-U_N)}\right)} \tag{8}$$

where $k = \max(\text{integer } i | (i+1)P - E < p*)$, $U_N = \sum_{\tau_i \in W} \frac{e_i}{p_i}$, and $p*$ represents the smallest period in W,

PROOF. Let us consider a component $C(W, R, \texttt{RM})$, where $W = \{\tau_i(p_i, e_i, b_i, m_i) | i = 1, \ldots, n\}$ and its periodic interface defined as $I(P, E, B, M)$. Let us assume the component in a normal non-faulty execution mode with a resource supply model $R = \Gamma(\Pi, \Theta, \Delta)$. According to work by Shin and Lee [16], the utilization bound of the component C under normal execution mode is given by

$$UB_W(\texttt{RM}) = U_I \cdot n \left[\left(\frac{2k + 2(1 - U_I)}{k + 2(1 - U_I)} \right)^{1/n} - 1 \right]$$

with k defined by $k = \max(\text{integer } i \mid (i + 1)P - E < p^*)$ and $U_I = \frac{E}{P}$.

In order to guarantee the schedulability of the component, the interface normal execution time E is the minimum value such that

$$U_N = \sum_{\tau_i \in W} \frac{e_i}{p_i} \leq U_I \cdot n \left[\left(\frac{2k + 2(1 - U_I)}{k + 2(1 - U_I)} \right)^{1/n} - 1 \right] \tag{9}$$

When n becomes large, we have

$$n \left[\left(\frac{2k + 2(1 - U_I)}{k + 2(1 - U_I)} \right)^{1/n} - 1 \right] \approx \log \left(\frac{2k + 2(1 - U_I)}{k + 2(1 - U_I)} \right) \tag{10}$$

Therefore, from Equations 9 and 10, it follows that

$$U_I \geq \frac{U_N}{\log\left(\frac{2k+2(1-U_I)}{k+2(1-U_I)}\right)}$$

Since $U_N \leq U_I$, we have

$$\log \left(\frac{2k + 2(1 - U_N)}{k + 2(1 - U_N)} \right) \leq \log \left(\frac{2k + 2(1 - U_I)}{k + 2(1 - U_I)} \right) \tag{11}$$

Equation (11) finally implies that

$$U_I \geq \frac{U_N}{\log\left(\frac{2k+2(1-U_N)}{k+2(1-U_N)}\right)}$$

Therefore, the minimum value for E is given by

$$E = \frac{P \cdot U_N}{\log\left(\frac{2k+2(1-U_N)}{k+2(1-U_N)}\right)}$$

However, since $U_N \leq 1$, it is easy to see that $E \leq P$. □

When a fault occurs on a task τ_i the resource utilization of the component increases by b_i/p_i and consequently, the resource supply to the component is also increased by Δ. Thus, the total resource utilization U_F of a component in presence of a fault is given by the following equation:

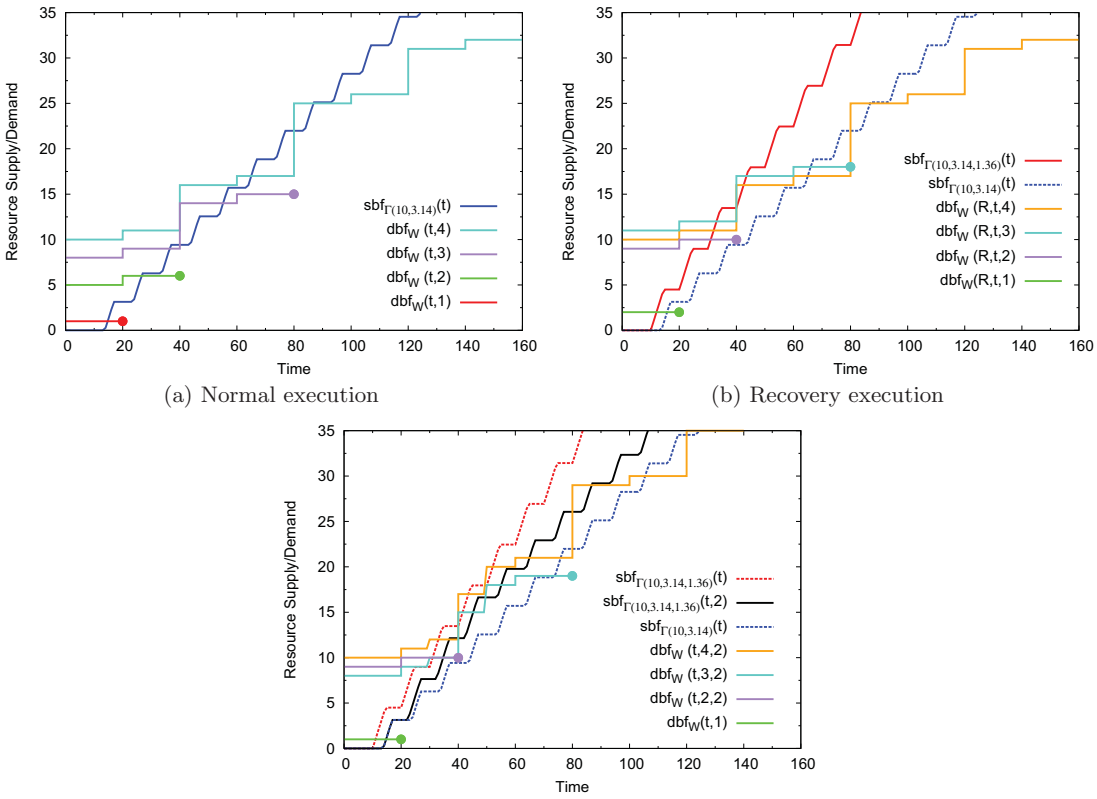
$$U_F = \sum_{\tau_i \in W} \frac{e_i}{p_i} + \max_{\tau_k \in W} \left(\frac{m_k b_k}{p_k} \right) \tag{12}$$

Since we assume only a single fault model, the value of the interface backup execution time B can be obtained by extending the result of Proposition 1 as given in Proposition 2

PROPOSITION 2. *The schedulability of a given component $C(W, R, \texttt{RM})$ abstracted by the interface $I(P, E, B, M)$, where $W = \{\tau_i(p_i, e_i, b_i, m_i) | i = 1, \ldots, n\}$ and $R = \Gamma(\Pi, \Theta, \Delta)$, is guaranteed if*

$$E + B = \frac{P \cdot U_F}{\log\left(\frac{2k+2(1-U_F)}{k+2(1-U_F)}\right)} \tag{13}$$

where $k = \max(\text{integer } i | (i + 1)P - (E + B) < p*$, and $p*$ represents the smallest period in W,



(a) Normal execution

(b) Recovery execution

(c) Fault analysis with $k = 2$

Figure 6: Schedulability analysis of Example 4.1

PROOF. The proof to Proposition 2 directly follows from that of Proposition 1. □

Finally, we determine M to be the maximum number of times the additional resource B is to be requested by the faulty component from its upper-layer component in case of fault. However, the value M should be decided to guarantee that the length of the resource supply cannot violate the deadline requirement of the faulty task, and that the total additional resource supplied for backup is large enough to cover the backup requirement of each task in case of fault. These two conditions are then formalized into Equation (14).

$$\begin{aligned} M \times P &\leq m_i \times p_i, \forall \tau_i \in W \\ M \times B &\geq m_i \times b_i, \forall \tau_i \in W \end{aligned} \tag{14}$$

From Equation (14), it follows that

$$\frac{m_i b_i}{B} \leq M \leq \frac{m_i p_i}{P}, \forall \tau_i \in W \tag{15}$$

We can still preserve the schedulability of the component by choosing M as the maximum value among the lower bound values that satisfy Equation (15) as shown in Equation (16).

$$M = \max_{\tau_i \in W} \left(\left\lceil \frac{m_i b_i}{B} \right\rceil \right) \tag{16}$$

Once the interface $I(P, E, B, M)$ of a component $C(W, R, A)$ is determined, the resource model $R = \Gamma(\Pi, \Theta, \Delta)$ provided by the upper-layer component to C can directly be derived from the interface by setting $\Pi = P$, $\Theta = E$, and $\Delta = B$.

EXAMPLE 4.1. *Let us consider a component* $C(W, R, \texttt{RM})$ *with its workload given by* $W = \{\tau_1(20, 1, 1, 1), \tau_2(40, 4, 4, 1), \tau_3(80, 3, 2, 1), \tau_4(160, 2, 0, 1)\}$. *Let us also assume that* T_F *is equal to 160, the least common multiple of all task periods in* W. *Now, let us set the period of the interface as* $P = 10$. *By choosing* $k = \lfloor \frac{p*}{P} \rfloor$ *in Proposition 1, we can obtain that* $E = 3.14$. *Similarly, we can obtain from Proposition 2 that* $B = 1.36$. *Equation (16) then provides* $M = 3$. *The component interface can then be given as* $I(10, 3.14, 1.36, 3)$ *and the resource model as* $R = \Gamma(10, 3.14, 1.36)$. *Thus, when a fault occurs in the component, an additional resource of* 1.36 *time units will be supplied to the components for* 3 *periods. Figure 6 shows the schedulability analysis of the component. Figure 6(a) tells that the component is schedulable under normal execution mode with the resource supply of* $\Gamma(10, 3.14)$. *However, in case of a fault as seen in Figure 6(b), the task* τ_2 *will miss its deadline with the resource supply of* $\Gamma(10, 3.14)$, *but the workload will preserve its schedulability with the resource supply of* $\Gamma(10, 3.14, 1.36)$. *We assumed a case where a fault is detected on* τ_2 *during the second resource supply. As seen in Figure 6(c), the schedulability of the workload is guaranteed by the resource model* $\Gamma(10, 3.14, 1.36)$ *which provided an extra* 1.36 *time units during* 3 *more resource intervals to backup the faulty task* τ_2.

5. CONCLUSION

This paper presents a new fault-tolerant compositional real-time scheduling framework. In the framework, each component contains a fault manager module which is in charge of detecting faults inside the component and recovering the faulty task by launching an associated backup strategy. The release of a backup task immediately increases the resource demand of a component. Thus, we introduced a fault-aware interface model to expose both the deadline and the fault requirements of each component to each upper-layer component. Furthermore, we provided a new fault-tolerant resource model that guarantees a minimum resource supply to a component in its normal execution mode, and increases the resource supply when the resource demand of the component increases due to a fault. Moreover, the resource also switches back to its minimum supply once the component has entirely recovered from the fault. We analyzed the schedulability of the new framework considering the Rate Monotonic scheduling algorithm and showed its efficiency over existing models.

In this paper, we considered only the task level fault management. Our future interest will be on a component and system levels fault management strategies. It will also be interesting to extend the fault model to for example a multiple fault model, since the occurrence of faults can be bursty or memoryless. Finally, we are planning to implement the framework on a real hardware to support the design and development of safety-critical avionics mission computers based on the ARINC-653 standard.

6. ACKNOWLEDGMENTS

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. NRF-2012R1A1A1015096), and the BK21 Plus Program (Research Team for Software Platform on Unmanned Aerial Vehicle, 21A20131600012) funded by the Ministry of Education (MOE, Korea) and National Research Foundation of Korea (NRF).

7. REFERENCES

[1] ARINC. Avionics application software standard interface: Part 1 - required services (arinc specification 653-2). Technical report, Aeronautical Radio, Incorporated, March 2006.

[2] M. Asberg, M. Behnam, F. Nemati, and T. Nolte. Towards hierarchical scheduling in autosar. In *Emerging Technologies Factory Automation*, ETFA 2009, pages 1–8, Sept 2009.

[3] S. Chen, L. T. X. Phan, J. Lee, I. Lee, and O. Sokolsky. Removing abstraction overhead in the composition of hierarchical real-time systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 81–90. IEEE Computer Society, 2011.

[4] A. Cunei and J. Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 68–77. ACM, 2006.

[5] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal. A compositional scheduling framework for digital avionics systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 371–380, August 2009.

[6] P. Eles, V. Izosimov, P. Pop, and Z. Peng. Synthesis of fault-tolerant embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1117–1122. ACM, 2008.

[7] M. A. Haque, H. Aydin, and D. Zhu. Real-time scheduling under fault bursts with multiple recovery strategy. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '14, pages –. IEEE Computer Society, 2014.

[8] J. Hyun and K. H. Kim. Fault-tolerant scheduling in hierarchical real-time scheduling framework. In *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '12, pages 431–436. IEEE Computer Society, 2012.

[9] J. Hyun, S. Lim, Y. Park, K. S. Yoon, J. H. Park, B. M. Hwang, and K. H. Kim. A fault-tolerant temporal partitioning scheme for safety-critical mission computers. In *Proceedings of the 31st IEEE/AIAA Digital Avionics Systems Conference*, DASC'12, pages 6C3–1–6C3–8. IEEE Computer Society, Oct 2012.

[10] H.-W. Jin. Fault-tolerant hierarchical real-time scheduling with backup partitions on single processor. *SIGBED Rev.*, 10(4):25–28, Dec. 2013.

[11] F. Many and D. Doose. Scheduling analysis under fault bursts. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 113–122. IEEE Computer Society, 2011.

[12] V. Mikolasek and H. Kopetz. Roll-forward recovery with state estimation. In *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '11, pages 179–186. IEEE Computer Society, March 2011.

[13] D. Nikolov, U. Ingelsson, V. Singh, and E. Larsson. Evaluation of level of confidence and optimization of roll-back recovery with checkpointing for real-time systems. *Microelectronics Reliability*, 54(5):1022–1049, 2014.

[14] R. M. Pathan and J. Jonsson. Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks. In *Proceedings of the 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '10, pages 265–274. IEEE Computer Society, 2010.

[15] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 237–246. IEEE Computer Society, 2013.

[16] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):30:1–30:39, April 2008.

[17] G. M. Tchamgoue, K. H. Kim, Y.-K. Jun, and W. Y. Lee. Compositional real-time scheduling framework for periodic reward-based task model. *Journal of Systems and Software*, 86(6):1712–1724, 2013.

[18] J. Xu and B. Randell. Roll-forward error recovery in embedded real-time systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 414–421. IEEE, June 1996.

[19] Y. Zhang and K. Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 320–327. IEEE, 2003.

Designing a Time-Predictable Memory Hierarchy for Single-Path Code

Bekim Cilku
Institute of Computer Engineering
Vienna University of technology
A 1040 Wien, Austria
bekim@vmars.tuwien.ac.at

Peter Puschner
Institute of Computer Engineering
Vienna University of technology
A 1040 Wien, Austria
peter@vmars.tuwien.ac.at

ABSTRACT

Trustable Worst-Case Execution-Time (WCET) bounds are a necessary component for the construction and verification of hard real-time computer systems. Deriving such bounds for contemporary hardware/software systems is a complex task. The single-path conversion overcomes this difficulty by transforming all unpredictable branch alternatives in the code to a sequential code structure with a single execution trace. However, the simpler code structure and analysis of single-path code comes at the cost of a longer execution time. In this paper we address the problem of the execution performance of single-path code. We propose a new instuction-prefetch scheme and cache organization that utilize the "knowledge of the future" properties of single-path code to reduce the main memory access latency and the number of cache misses, thus speeding up the execution of single-path programs.

Keywords

hard real-time systems, time predictability, memory hierarchy, prefetching, cache memories

1. INTRODUCTION

Embedded real-time systems need safe and tight estimations of the Worst Case Execution Time (WCET) of time-critical tasks in order to guarantee that the deadlines imposed by the system requirements are meet. Missing a single deadline in such a system can lead to catastrophic consequences.

Unfortunately, the process of calculating the WCET bound for contemporary computer systems is, in general, a complex undertaking. On the one hand, the software is written to execute fast – it is programmed to follow different execution paths for different input data. Those different paths, in general, have different timing, and analyzing them all can lead to cases where the analysis cannot produce results of the desired quality. On the other hand, the inclusion of hardware features (cache, branch prediction, out-of-order execution, and pipelines) extend the analysis with state dependencies and mutual interferences; a high-quality WCET analysis has to consider the interferences of all mentioned hardware features to obtain tight timing analysis. The state-of-the-art tools for WCET analysis are using a highly integrated approach by considering all interferences caused by hardware state interdependencies [4]. Keeping track of all possible interferences and also the hardware state history for the whole code in an integrated analysis can lead to a state-space explosion and will make the analysis infeasible. An effective approach that would allow the tool to decompose the timing analysis into compositional components is still lacking [1].

One strategy to avoid the complexity of the WCET analysis is the single-path conversion [12]. The single-path conversion reduces the complexity of timing analysis by converting all input-depended alternatives of the code into pieces of sequential code. This, in turn, eliminates control-flow induced variations in execution time. The benefit of this conversion are the predictable properties that are gained with the code transformation. The new generated code has a single execution trace that forces the execution time to become constant. To obtain information about the timing of the code it is sufficient to run the code only once and to identify the stream of the code execution which is repeated on any other iteration.

Large programs that have been converted into single-path code can be decomposed into smaller segments where each segment can be easily analyzed for its worst-case timing in separation. This contrasts the analysis of traditional code, where a decomposition into segments may lead to highly pessimistic timing-analysis results, because important information about possible execution paths and information about how these execution paths within one segment influence the feasible execution paths and timings in subsequent segments gets lost at segment boundaries. In single-path code, each code segment has a constant trace of execution and the initial hardware states for each segment can be easily calculated, because there are no different alternatives of the incoming paths that can lead to a loss of information during a (de)compositional analysis. However, the advantage of generating compositional code that allows for a highly accurate, low-complexity analysis comes at the cost of a longer execution time of the code.

The long latency of memory accesses is one of the key performance bottlenecks of contemporary computer systems. While the inclusion of an instruction cache is a crucial first step to bridge the speed gap between CPU and main memory, this is still not a complete solution – cache misses can result in significant performance losses by stalling the CPU until the needed instructions are brought into the cache.

For such a problem, prefetching has been shown to be an effective solution. Prefetching can mask large memory latencies by loading the instructions into the cache before they are actually needed [15]. However, to take advantage of this improvement, the prefetching commands have to be issued at the right times – if they are issued too late memory latencies are only partially masked, if they are issued too early, there is the risk that the prefetched instruction will

evict other useful instructions from the cache.

Prefetching mechanisms also have to consider the accuracy, since speculative prefetching may pollute the cache. Mainly the prefetching algorithms can be divided into two categories: correlated and non-correlated prefetching. Correlated prefetching associates each cache miss with some predefined target stored in a dedicated table [6, 16], while non-correlated ones predict the next prefetch line according to some simple predefined algorithms [11, 7, 14].

For all mentioned techniques, the ability to guess the next reference is not fully accurate and prefetching can result in cache pollution and unnecessary memory traffic. In this paper we propose a new memory hierarchy for single-path code that consists of a cache and a hardware prefetcher. The proposed design is able to prefetch sequential and non-sequential streams of instructions with full accuracy in the value and time domain. This constitutes an effective instruction prefetching scheme that increases the execution performace of single-path code and reduces both cache pollution and useless memory traffic.

The rest of the paper is organized as follows. Section 2 gives a short description of predicated instruction and presents some simple rules used to convert conventional code to single-path code. The new proposed memory hierarchy is presented in Section 3. Section 4 discusses related work. Finally, we make concluding remarks and present the future work in Section 5.

2. GENERATING SINGLE-PATH CODE

The goal of the single-path code-generation strategy is to eliminate the complexity of multi-path code analysis, by eliminating branch instructions from the control flow of the code. Different paths of program execution are the result of branch instructions which force the execution to follow different sequences of instructions. Branch instructions can be unconditional branches which always result in branching, or conditional branches where the decision for the execution direction depends on the evaluation of the branching condition.

The single-path conversion transforms conditional branches, i.e., those branches whose outcome is influenced by program inputs [12]. Before the actual single-path code conversion is done, a data-flow analysis [3] is run to identify the input-dependent instructions of the code. Branches which are not influenced by the input values are not affected by the transformation. After the data-flow analysis, the single-path conversion rules are applied and the new single-path code is generated. The only additional requirement for executing single-path converted code is that the hardware must support the execution of predicated instructions.

2.1 Predicated execution

Predicated instructions are instructions whose semantics are controlled by a predicate (or guard), where the predicate can be implemented by a specific predicate flag or register in the processor. Instructions whose predicate evaluate to "true" at runtime are executed normally, while those which evaluate to "false" are nullified to prevent that the processor state gets modified.

Predicated execution is used to remove all branching operations by merging all blocks into a single one with straight-line code [10]. For architectures that support predicated (guarded) execution the compiler converts conditional branches into (a) predicate-defining instructions and (b) sequences of predicated instructions – the instructions along the alternative paths of each branch are converted into sequences of predicated instructions with different predicates.

```
if(a)              beq    a,0,L1      pred_eq  p,a
  x=x+1            add    x,x,1       add      x,x,1 (p)
else               jump   L2          add      y,y,1 (not p)
  y=y+1        L1:
               add    y,y,1
             L2:
```

Figure 1: if-conversion

Figure 1 shows an example of an *if-then-else* structure translated in assembler code with and without predicated instructions. In the first assembler code, depending on the outcome of the branch instruction, only part of the code will be executed, while in the second, single-path case all instruction will be executed but the state of the processor will be changed only for instructions with true predicated value.

2.2 Single-Path Conversion Rules

In the following we describe a set of rules to convert regular code into a single-path code [13]. Table 1 shows the single-path transformation rules for sequences, alternatives and loops structures. In this table we assume that conditions for alternatives and loops are simplified in boolean variables. The precondition for statement execution is represented with σ, while in cases of recursion the δ counter is used to generate unique variable name.

Simple Statement. If precondition for simple statement S is always true then the statement will be executed in every execution. Otherwise the execution of S will depend on the value of the precondition σ, which becomes the execution predicate. The same rule is used for statement sequences, by applying the rule sequentially to each part of the sequence.

Conditional Statement. For input-dependent (*ID(cond)* is *true*) branching structures, we serialize the S_1 and S_2 alternatives, where the precondition parameters of the alternatives $S_1$and S_2 are formed by a conjunction of the old precondition (σ) and the outcome of the branching condition that is stored in $guard_\delta$. If branching is not dependent on program inputs then the *if-then-else* structure is conserved and the set of rules for single-path conversion are applied individually to S_1 and S_2.

Loop. Input-data dependent loops are transformed in two steps. First, the original loop is transformed into a *for-loop* and the number of iterations N is assigned – the iteration count N of the new loop is set to the maximum number of iterations of the original loop code. The termination of the new new loop is controlled by a new counter variable ($count_\delta$) in order to force the loop to iterate always for the constant number N. Further, a variable end_δ is introduced. This variable is used to enforce that the transformed loop has the same semantics as the original one. The end_δ-flag stored in this variable is initialised to $true$ and assumes the value $false$ as soon as the termination condition of the original loop evaluates to $true$ for the first time. The value of end_δ-flag can also be changed to $false$ if a break is embedded into the loop. Thus S is executed under the same condition as in the original loop.

Table 1: Single-Path Transfromation Rules

Construct S		Translated Construct $\mathcal{SP}[\![\, S \,]\!]\, \sigma\delta$
S	if $\sigma = T$	S
	otherwise (σ)	S
$S_1\,;S_2$		$\mathcal{SP}[\![\, S_1 \,]\!]\, \sigma\delta\,;$ $\mathcal{SP}[\![\, S_2 \,]\!]\, \sigma\delta$
`if` *cond* `then` S_1 `else` S_2	if *ID(cond)*	$guard_\delta$ `:=` *cond*`;` $\mathcal{SP}[\![\, S_1 \,]\!]\langle\sigma \wedge \ \ guard_\delta\rangle\langle\delta + 1\rangle\,;$ $\mathcal{SP}[\![\, S_2 \,]\!]\langle\sigma \wedge \neg guard_\delta\rangle\langle\delta + 1\rangle$
	otherwise	`if` *cond* `then` $\mathcal{SP}[\![\, S_1 \,]\!]\, \sigma\delta$ `else` $\mathcal{SP}[\![\, S_2 \,]\!]\, \sigma\delta$

`while` *cond*
`max` N `times`
`do` S
	if *ID(cond)*	end_δ `:= false`
		`for` $count_\delta$ `:=` 1 `to` N `do begin`
		$\mathcal{SP}[\![$ `if` $\neg$*cond* `then` end_δ `:= true` $]\!]\, \sigma\langle\delta + 1\rangle\,;$
		$\mathcal{SP}[\![$ `if` $\neg end_\delta$ `then` S $]\!]\, \sigma\langle\delta + 1\rangle$
		`end`

	otherwise	`while` *cond* `do` $\mathcal{SP}[\![\, S \,]\!]\, \sigma\delta$

3. MEMORY HIERARCHY FOR SINGLE-PATH CODE

This section presents our novel architecture of the cache memory and the prefetcher used for single-path code.

3.1 Architecture of the Cache Memory

Caches are small and fast memories that are used to improve the performance between processors and main memories based on the principle of locality. The property of locality can be observed from the aspects of temporal and spatial behavior of the execution. Temporal locality means that the code that is executed at the moment is likely to be referenced again in the near future. This type of behavior is expected from program loops in which both data and instructions are reused. Spatial locality means that the instructions and data whose addresses are close by will tend to be referenced in temporal proximity because the instructions are mostly executed sequentially and related data are usually stored together [15].

As an application is executed over the time, the CPU makes references to the memory by sending the addresses. At each such step, the cache compares the address with tags from the cache. References (instructions or data) that are found in cache are called *hits*, while those that are not in the cache are called *misses*. Usually the processor stalls in case of cache misses until the instructions/data have been fetched from main memory.

Figure 2 shows an overview of the cache memory augmented with the single-path prefetcher. The cache has two banks, each consisting of *tag*, *data*, and *valid bit* (V) entries. Separation of the cache into two banks allows us to overlap the process of fetching (by the CPU) with prefetching (by the prefetch unit) and also cost less than dual-port cache of the same size. At any time, one of the banks is used to send instructions to the CPU and the other one to prefetch instructions from the main memory. Both, CPU and prefetcher can issue requests to the cache memory. Whenever a new value in program counter (PC) is generated the value is sent to the cache and to the prefetcher. There are three different cases of cache accesses when the CPU issues an instruction request:

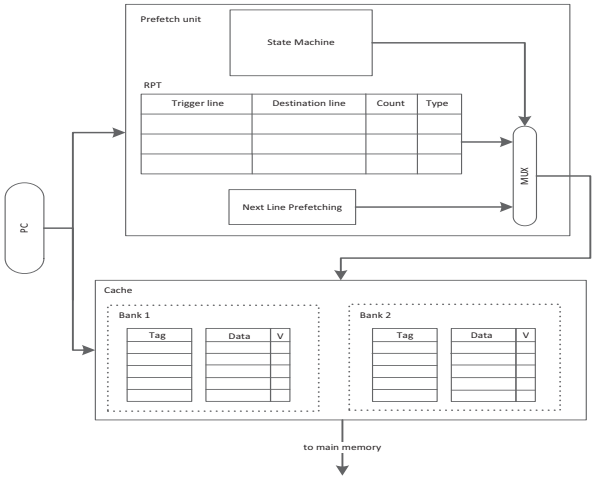


Figure 2: Prefetch-Cache architecture

- No match within tag columns - the instruction is not in the cache. The cache stalls the processor and forwards the address request to the main memory;
- Tag match, V bit is zero - the instruction is not in the cache but the prefetcher has already sent the request for that cache line and the fetching is in progress. The cache stalls the processor and waits for the ongoing prefetching operation to be finished (V value to switch from zero to one).

- Tag match, V bit is one - the instruction is already in the cache (cache hit).

The V bit prevents the cache to replicate requests issued to main memory if the same request has already been issued by the prefetcher.

Instructions in the cache can be mapped to any location (fully associative), to a dedicated set of cache lines (set-associative) or to only one cache line location (direct-mapped). For single-path code, direct mapped is the most conventional cache solution. In single-path code, loop have sequential bodies and if the loop size is $cache_{size} < loop_{size} < 2 * cache_{size}$ then a minimal number of instructions will be evicted from the cache. To illustrate this, let us assume that a cache has a size of four cache lines $(1, 2, 3, 4)$ and the loop has a size equal to six cache lines (a, b, c, d, e, f). In each iteration the loop will be mapped in cache as: $(a, e) \rightarrow 1$, $(b, f) \rightarrow 2$, $(c) \rightarrow 3$, and $(d) \rightarrow 4$. In each iteration lines (c, d) are in the cache.

3.2 Prefetching Algorithm for Single-Path Code

The prefetching algorithm for single-path code considers two forms of prefetching: sequential and non-sequetial prefetching. Sequential instruction streams are a trivial pattern to predict since the address of the next prefetching is an increment of the current address line. A simple next-line prefetcher [14] is a suitable solution for such a pattern of instructions.

In contrast, a non-sequential prefetcher needs input information to determine the target address to be prefetched. Single-path code has a strong advantage in this part of the prefetching, because the outcome of every branch target is statically known. This target information can be given to the prefetcher in form of instructions (software prefetching) or it can be kept in a local memory (organized as a table) and used by the prefetcher when it is needed (hardware prefetching). For the software solution, special prefetch instructions are needed and the CPU hardware has to be modified.

In order to keep the development of the prefetcher independent form the CPU and the compiler, and also to avoid the overhead of executing fetch instructions in software, we have decided to use a hardware solution for the single-path prefetcher. Since the single-path code consists of serial segments and loops only, the subject of treatment from non-sequential algorithm are only the branch instructions of the loop back-edge. Loops larger than a cache size are easily handled by the prefetching algorithm, where the loop body is prefetched with the sequential algorithm while for the loop header with non-sequential one. If loops fully fit into the cache then they do not need to be prefetched on each iteration. Thus these loops are identified and the prefetcher does not generate any prefetching requests until the last iteration, when the execution stream exits the loop.

The granularity of prefetching is defined as one cache line. For larger amounts of prefetched instructions, the probability of overshoting the end of sequence would increase, thus resulting in cache pollution with useless prefetching. The granularity also determines that the prefetching distance is one cache line ahead.

3.3 Reference Prediction Table

The Reference Prediction Table (RPT) is the part of the prefetcher that holds information about the instruction stream (Figure 2). The RPT entries consist of *trigger address, destination address, count* and *type* column. *Trigger address* is the program counter address that triggers the non-sequential algorithm of the prefetcher. *Destination address* is the target address that is prefetched. Since loops in single-path code have a constant number of iterations the *counter* data is used to inform the prefetcher for how many times the target address should be prefetched. The *type* field indicates which loops fit into the cache and which loops are bigger than the cache. If the value of *type* is zero then the prefetcher will not take any action since the loop is smaller than cache and is completely in it. When the *counter* of that loop reaches zero, the loop iterations are finished, and the prefetcher triggers the prefetching of the next cache line.

A profiling process is needed to identify loops (loop header and back-edge branch of the loop), the number of iterations and the size of the loops in order to fill the RPT table.

3.4 Architecture of the Prefetcher

As shown in Figure 2 the prefetch hardware for single-path code consist of the Reference Prediction Table (RPT), the next-line prefetcher, and the prefetch controller (state machine). The next-line prefetcher serves for prefetching the sequential parts of the code, while the state machine in association with the RPT is used for pefetching targets in distance.

At run-time, when a new address is generated, its value is passed to the RPT table and the next-line prefetcher. In cases when the PC value matches an entry in the RPT table, the prefetch controller reads the *type* bit and the *counter* value to check if the loop is smaller/bigger than cache and on each interation if the final iteration has been reached. If there is no match with the RPT table entry, the next-line prefetching will increment the address for one cache line and issue that address to the cache. The RPT output has precedence over next-line prefetching.

4. RELATED WORK

Designers have proposed several strategies to increase the performance of cache-memory systems. Some of these approaches use software support to perform prefetching, while others are strictly hardware-based. Software solutions need explicit fetch instructions to be issued from the compiler to do prefetching. In this section we discuss only related hardware solutions.

The simplest form of prefetching comes passively from the cache itself. When a cache miss occurs, besides the missed instruction the cache fetches also instructions that belong to the same line into the cache. An extention of the cache line size implies a larger granularity – more instructions are fetched on a cache miss. The disadvantages are that longer cache line take longer to fill, generate useless memory traffic, and also they contribute to cache pollution [5].

The "one block look ahead" prefetching, later extended to "next-N-line", prefetches cache lines that are following the one currently being fetched by the CPU [14]. The scheme requires small additional hardware to compute the next sequential addresses. Unfortunately, "next-n-line" is unlikely to improve performance when execution proceeds through non-sequential execution paths. In this case the prefetching guess can be incorrect.

Tagged prefetching has a tag bit associated with each cache line [7]. When a line is prefetched, its tag bit is set

to zero. If the line is used then the bit is set to one and the next sequential line is immediately prefetched. The stream buffer is a similar approach except that the buffer stands between main memory and cache in order to avoid polluting the cache with data that may never be needed.

The target prefetching scheme addresses the problem of non sequential code [16]. This approach comprises a next-line prediction table consisting of two entries (current line address and target line address). When the program counter changes the value, the prefetcher searches the prediction table. If there is a match, then the target address is a candidate for prefetching.

The hybrid scheme that combines both next-line and target prefetching offers a cumulative accuracy for reducing cache misses. However, this solution has limited effectiveness since the predicted direction is defined from the previous execution. A similar approach is also used on "wrong-path" prefetching except that instead of target table this approach prefetches immediately the target of conditional branch after the branch instruction is recognized in in the decode stage [11]. This solution can be effective only for non-taken branches. The Markov prefetcher [6] prefetches multiple reference predictions from the memory subsystem, by observing the miss-reference stream as an Markov model.

Loop-based instruction prefetcing [2] is similar to our solution, except that the loop headers are always prefetched and the prefetching is issued at the end of the loop with no prefetch distance. The cooperative approach [9] also considers sequential and non-sequential prefetching by using a software solution for non-sequential prefetch. A dual-mode instruction prefetch scheme [8] is an alternative to improve worst-case execution time by associating a thread to each instruction block that is part of WCET. Threads are generated by the compiler and they are static during task execution.

5. CONCLUSION AND FUTURE WORK

To overcome the problem of long execution times of single-path code, we have proposed a new memory hierarchy organization that attempts to reduce the memory access time by bringing the instructions into the cache before they are required.

The single-path prefetching algorithm combines a sequential and a non-sequential prefetching scheme with the full accuracy in the predicted instruction stream based on the predictable properties of the single-path code. Designed as a hardware solution, the prefetcher does not produce an additional timing overhead for the instruction prefetching. Also, our solution allows the prefetcher functionality to be independent without interfering with any stage of CPU. The dual-bank cache makes it possible to pipeline the CPU and prefetcher accesses into the cache memory in order to fully utilize the memory bandwidth. By using a prefetch granularity of one cache line we eliminate the possibility for cache pollution and useless memory traffic.

In our future work we plan to show the feasibility of the memory hierarchy by implementing it in an FPGA platform and also to extend the prefetcher for input-independent if-else structures that are not converted to sequential code.

Acknowledgments

This work has been supported in part by the European Community's Seventh Framework Programme [FP7] under grant agreement 287702 (MultiPARTES) and the EU COST Action IC1202: Timing Analysis on Code Level (TACLe).

6. REFERENCES

[1] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[2] Y. Ding and W. Zhang. Loop-based instruction prefetching to reduce the worst-case execution time. *Computers, IEEE Transactions on*, 59(6):855–864, 2010.

[3] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Code analysis for temporal predictability. *Real-Time Systems*, 32(3):253–277, 2006.

[4] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis-definition and challenges. In *6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.

[5] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[6] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[7] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373. IEEE, 1990.

[8] M. Lee, S. L. Min, C. Y. Park, Y. H. Bae, H. Shin, and C. S. Kim. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 98–105. IEEE, 1993.

[9] C.-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, (1):71–109, 2001.

[10] J. C. Park and M. Schlansker. On predicated execution. Technical report, Technical Report HPL-91-58, HP Labs, 1991.

[11] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 165–175. IEEE, 1996.

[12] P. Puschner and A. Burns. Writing temporally predictable code. In *Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on*, pages 85–91. IEEE, 2002.

[13] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. Compiling for time predictability. In *Computer Safety, Reliability, and Security*, pages 382–391. Springer, 2012.

[14] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[15] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[16] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597. IEEE Computer Society Press, 1992.

Five problems in compositionality of real-time systems

Björn Andersson
Carnegie Mellon University

ABSTRACT

This abstract presents five important problems in compositionality of real-time systems.

Statement

The general problem of compositionality of real-time systems is to create a concept which describes resource consumption of something so that this concept can be taken as input to analysis. In this abstract, I present a list of five problems of compositionality that I believe are important:

1. Create a provably good interface for constrained-deadline sporadic tasks on a single processor. Many previously proposed interfaces are based on bandwidth-like quantities; this makes them unable to achieve provably good performance as seen from this example: Consider a system with a single processor and two components. Component 1 comprises k tasks $(\tau_1, \tau_2, \ldots, \tau_k)$ characterized by $T_i = \infty, D_i = i, C_i = 1$. Component 2 comprises one tasks (τ_{k+1}) characterized by $T_{k+1} = \infty, D_{k+1} = k+1, C_{k+1} = 1$. For each value of $k \geq 1$, if tasks were scheduled directly on the processor using EDF then the taskset is schedulable. Using bandwidth-like interfaces, however, we obtain that component 1 requires the bandwidth $\sum_{i=1}^{k}(1/i)$ and component 2 requires the bandwidth $(1/(k+1))$. If such an interface would be used and if a schedulability test would take these bandwidths as input then, for $k \geq 2$, the bandwidth required exceeds 100% and hence the taskset would be deemed unschedulable. For $k \to \infty$, the required bandwidth (as stated by the interfaces) would be infinite. Hence, there are tasksets that are schedulable directly on the processor but if a bandwidth-like interface is used, even an infinite speedup of the processor is not enough to make the taskset schedulable.

2. Create a concept that describes the memory accesses of a task. This concept should describe the possible timing and the possible memory addresses and it should be possible to take this as input to timing analysis that is aware of contention for resources in the memory system.

3. Create a concept that describes the aggregate resource consumption of a complex interaction in a protocol. This is particularly important for compositional analysis of real-time communication over shared communication medium of traffic that uses Transmission Control Protocol (TCP) because in such a setting, the acknowledgement packets consume network bandwidth but they are dependent on the data-carrying packets. So it is desirable to create a concept that describes the aggregate resource consumption of both data packets and ack packets of a given flow. On a higher level, this problem becomes important when using software frameworks for distributed systems that rely on TCP.

4. Create a concept that describes aggregate resource consumption of wireless traffic. On a wireless medium, a transmitted packet can get corrupted by noise requiring a retransmission. But then a concept that describes that aggregate resource consumption of wireless traffic of a given traffic flow needs to be a function of noise parameters of the channel.

5. Create a concept that describes the aggregate resource consumption of software in an autonomous car. Some tasks in an autonomous car are just like other types of embedded systems; they periodically read sensors, perform computations, and actuate commands. This is true for low-level control loops and some sensor fusion tasks in autonomous cars. But an autonomous car needs to perform other computations as well related to higher-level cognition and responding to events and planning future actions. And the resource consumption of such tasks depends on the environment. (For example, if you drive in a dense urban environment, there are more events to deal with than if you drive on a highway.) Hence, there is the need to find an "eventfulness" metric of a physical world and describe the resource consumption of the software as a function of this metric.

There should be an analysis that takes the concept as input and this analysis should have low pessimism and there should be a metric that characterizes this low pessimism and the concept should be capable of describing systems so that the description requires few bits because this indicates good information hiding.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001733

Compositional Mixed-Criticality Scheduling

Extended Abstract

Arvind Easwaran
Nanyang Technological University, Singapore
arvinde@ntu.edu.sg

Insik Shin
KAIST, Korea
insik.shin@cs.kaist.ac.kr

An increasingly important trend in embedded systems is towards open computing environments, where multiple functionalities are developed independently and integrated together on a single computing platform; this trend is evident in industry-driven initiatives such as ARINC653 in avionics and AUTOSAR in automotive. An important notion behind this trend is the safe isolation of separate functionalities, or *partitioning*, to achieve fault containment and compositional validation/certification. This raises the challenge of how to balance the conflicting requirements of partitioning for safety assurance and resource sharing for economical benefits. The concept of *mixed-criticality* appears to be important in meeting those two seemingly conflicting goals.

In many safety-critical embedded systems, the correct behavior of some functionality (e.g., flight control) is more important ("critical") to the overall safety of the system than that of another (e.g., in-flight entertainment). In order to certify such systems as being correct, they are conventionally assessed under certain assumptions on the worst-case runtime behavior. For example, the estimation of Worst-Case Execution Times (WCETs) of code for highly critical functionalities, typically involves very conservative assumptions that are unlikely to occur in practice [3]. Such assumptions make sure that the resources reserved for critical functionalities are always sufficient. Thus, the system can be designed to be safe from a validation and certification perspective, but the resources are in fact severely under-utilized in practice.

In order to close such a gap in resource utilization, Vestal [6] proposed the mixed-criticality (MC) task model that comprises different WCET values. These different values are determined at different levels of confidence ("criticality"), based on the following principle. A reasonable low-confidence WCET estimate, even if it is based on measurements, may be sufficient for almost all possible execution scenarios in practice. In the highly unlikely event that this estimate is violated, as long as the scheduling mechanism can ensure deadline satisfaction for highly critical applications, the resulting system design may still be considered as safe.

To ensure deadline satisfaction of critical applications, existing mixed-criticality studies make pessimistic assumptions when a single high-criticality task executes beyond its expected (low-confidence) WCET. They assume that the system will either immediately ignore all the low-criticality tasks (e.g., [1, 4]) or degrade the service offered to them (e.g., [2, 5]). They further assume that all the high-criticality tasks in the system can thereafter request for additional resources, up to their pessimistic (high-confidence) WCET estimates. Although these strategies ensure safe execution of critical applications, they have a serious drawback as pointed out in a recent article [2]. When a high-criticality task exceeds its expected WCET, the likelihood that all the other high-criticality tasks in the system will also require more resources is very low in practice. Therefore, to penalize all the low-criticality tasks for this unlikely event seems unreasonable.

Proposed Research.

We are currently exploring a new research direction using component-based mixed-criticality system model to address the above issues. Designer tunable component boundaries can be used to isolate low-criticality tasks from unexpected variations in high-criticality WCETs. Thus, it is possible to allow low-criticality tasks in some components to continue their execution uninterrupted, even when some high-criticality tasks in other components have exceeded their expected WCET. Some of the main challenges in this direction of research are as follows: 1) How to enable the designer to tune low-criticality isolation capabilities of a component, 2) What does a mixed-criticality component interface look like, 3) How does a component indicate criticality change to the rest of the system, 4) What impact does a criticality change inside a component have on the rest of the system, and 5) Can we maintain compositionality even when the impact of criticality changes are not confined within components.

1. REFERENCES

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, and A. Marchetti-Spaccamela. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *ECRTS*, 2012.

[2] A. Burns and S. Baruah. Towards a More Practical Model for Mixed-Criticality Systems. In *Workshop on Mixed-Criticality Systems (co-located with RTSS)*, 2013.

[3] A. Burns and B. Littlewood. Reasoning about the reliability of multi-version, diverse real-time systems. In *RTSS*, 2010.

[4] A. Easwaran. Demand-based Scheduling of Mixed-Criticality Sporadic Tasks on One Processor. In *RTSS*, 2013.

[5] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *ASP-DAC*, 2014.

[6] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *RTSS*, 2007.

Challenges of Virtualization in Many-Core Real-Time Systems

Matthias Becker, Mohammad Ashjaei, Moris Behnam, Thomas Nolte
MRTC / Mälardalen University, Västerås, Sweden
{matthias.becker, mohammad.ashjaei, moris.behnam, thomas.nolte}@mdh.se

ABSTRACT

Embedded real-time virtualization is used for single core and multicore platforms to consolidate multiple systems within a single chip. The number of cores on one processor is steadily increasing, making many-core processors with tens to hundreds of cores available in the near future. This gives rise to a number of new challenges for real-time virtualization on such systems. In this work we identify a number of those challenges. We also describe initial ideas on how to tackle them.

1. INTRODUCTION

Virtualization is a key component in todays industrial systems [4]. Many-core processors with tens to hundreds of cores, connected by a 2D-mesh based Network-on-Chip (NoC) bring new challenges. The large number of simple cores deliver enough resources to allocate whole cores to virtual partitions. Therefore, the new challenge will be how to divide the hardware topology in order to guarantee enough resources, computational and memory bandwidth, for all the guest systems. Hereby a guest system is abstracted by an interface, hence can be seen as a component. Such components describe newly developed parallel applications as well as legacy applications designed for single-core systems.

2. PROBLEM DESCRIPTION

An important challenge is the interface definition needed to abstract the resource requirement of guest systems. Given those interfaces, system integration needs to find a suitable composition of guest systems on the many-core. Runtime mechanisms are needed to police the resource access depending on the parameters defined during system integration.

2.1 Challenges for the guest interface

Each guest system has to operate under certain constraints. For many-cores, knowing the location of the cores relative to each other is important. This information is needed in order to perform schedulability analysis of the NoC communication [2]. On recent many-core processors (e.g. Tilera's Tile64 or Kalrays MPPA-256 processor [6, 3]) different NoCs are provided for core to core communication and memory access. Finding the right parameters and abstractions for the interface is an important challenge. Moreover, we may have dependencies among guests, hence guest communication abstraction is required. We propose to define those constraints by an interface: $G_n = \{\beta, N_x, N_y, \mathcal{L}\}$, where n is the index of the system and β is the required bandwidth to offchip memory. The parameters N_x and N_y specify the size and shape of the guest system (i.e. $N_x \times N_y$ yields the number of cores). The set $\mathcal{L}$ defines the individual communications to other guests. Evaluating the minimal values for the interface parameters is another important challenge.

2.2 Challenges for system integration

Since we consider real-time systems, we target static constellations. At design time, a set of guest systems, $\mathcal{G}$, needs to be mapped to the many-core hardware. Since this step is performed offline, sophisticated mapping techniques can be applied. Challenges for the mapping are multidimensional. A physical mapping to the cores needs to guarantee the required network bandwidth and timeliness of the guest communication. Communication latencies depend on the physical location of the cores.

2.3 Challenges for runtime mechanisms

At runtime, allocation of resources specified by a guest interface needs to be guaranteed. Computational resources are statically assigned, thus they do not need to be monitored. However access to the offchip memory is shared among guests. Therefore, distribution of the memory bandwidth during runtime is an important challenge. Lu et al. proposed the (σ, ρ)-based flow regulation for NoC [5] which itself is based on Network Calculus [1]. In order to regulate the outgoing traffic flows on a NoC link, a packet shaper is used to inject messages based on a given profile rather than as fast as possible. This mechanism is successfully implemented in the MPPA-256 processor [3]. We propose to include a software-based packet shaper in the hypervisor. Virtualization of the NoC for guest-to-guest communication may affect the communication inside a third guest as they use the same network. This raises another challenge which is to investigate a proper mechanism to handle communication on this network.

3. REFERENCES

[1] R. Cruz. A calculus for network delay. I. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[2] D. Dasari et. al. Noc contention analysis using a branch-and-prune algorithm. *ACM Trans. Embed. Comput. Syst.*, 13:113:1–113:26, 2014.

[3] B. D. de Dinechin et. al. Time-critical computing on a single-chip massively parallel processor. In *DATE '14*, pages 97:1–97:6, 2014.

[4] G. Heiser. The role of virtualization in embedded systems. In *IIES '08*, pages 11–16, 2008.

[5] Z. Lu et. al. Flow regulation for on-chip communication. In *DATE '09*, pages 578–581, 2009.

[6] Tilera. Tile64 processor. http://www.tilera.com/products/processors, Retrieved Oct. 30, 2014.

Managing end-to-end resource reservations

Extended Abstract

Luis Almeida[1,2]
[1]IT - University of Porto
Porto, Portugal
lda@fe.up.pt

Moris Behnam[2]
[2]IDT - Mälardalen University
Västerås, Sweden
moris.behnam@mdh.se

Paulo Pedreiras[3]
[3]IT - University of Aveiro
Aveiro, Portugal
pbrp@ua.pt

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications, Distributed networks*

Keywords

Computer networks, Resource reservations, Scalability, Adaptability

1. CONTEXT

Currently, there is a strong push for real-time applications distributed over large geographical areas, fuelled by interactive multimedia, remote interactions, cloud-based time-sensitive services and many cyber-physical systems [3]. These applications rely on many different resources, from the end nodes where they execute, to the networks over which they communicate. However, supporting global compositionality in this realm, i.e., guaranteeing applications performance a priori independently of load variations, is a significant challenge. It calls upon resource reservations across the system that comply to performance metrics agreed at the time of application deployment, what is normally called Service Level Agreement (SLA). This requires well defined application resource requirements, from the computing end nodes, possibly including multiple internal resources, to the network.

In spite of existing technical solutions for enforcing reservations, from traffic shaping in networks to virtualization in processors and reservations in static distributed systems [2], such solutions typically fall short on at least two aspects, scalability and adaptability. The former is needed to cope with large numbers of reservations to achieve the desired segregation and isolation among many applications, while the latter is needed to mitigate the effects of overprovisioning that micro-reservations can have in the efficiency of the whole system. These two requirements apply particularly to the network, given its central position in supporting the applications referred above and, in general, emerging paradigms such as the Internet-of-Things. Nevertheless, adaptability naturally extends from the network to the ends nodes, for the sake of efficiency, too.

On the other hand, the scalable solutions that we have today, e.g., deployed in current Internet, provide at best class-based Quality-of-Service (QoS) support [1]. This grants protection against interference from traffic of lower QoS delay tolerant classes but not within the same class, which is aggravated when the number of similar applications grows, e.g. VoIP calls.

Finally, supporting scalability and adaptability requires agile resource management. If the former points to a distributed management architecture, the latter is better achieved with a centralized one. This apparent conflict is typically dealt with using clustering. Local reservations are managed at the cluster level while end-to-end reservations require a distributed cluster-level protocol. An example of such approach is the Stream Reservation Protocol (SRP) in Audio-Video Bridges, but it is still class-oriented.

2. OPEN PROBLEMS

Therefore, given the considerations above, we herein formulate what we consider to be open problems to achieve the desired compositionality of distributed real-time applications in large open systems, in a resource efficient way. Given a distributed real-time application that will execute in N end nodes connected to a large network, how to:

- Formulate its resource requirements and interfaces?
- Express adaptivity in such requirements/ interfaces?
- Support scalable and adaptive network reservations?
- Analyze the requirements feasibility?
- Carry out global admission control and enforce the needed reservations?
- Track and distribute slack?

3. ACKNOWLEDGMENTS

With support from the Portuguese Gov. through FCT grants CodeStream (PTDC/EEI-TEL/3006/2012) and ServCPS (PTDC/EEAAUT/122362/2010).

4. REFERENCES

[1] G. Bertrand, S. Lahoud, M. Molnar, and G. Texier. Qos routing and management in backbone networks. In *Intell. QoS Tech. and Network Manag.t: Models for Enhancing Comm.*, pages 138–159. IGI Global, 2010.

[2] N. Serreli, G. Lipari, and E. Bini. Deadline assignment for component-based analysis of real-time transactions. In *CRTS 2009 Proceedings*, December 2009.

[3] F. Xia, L. Ma, J. Dong, and Y. Sun. Network qos management in cyber-physical systems. In *ICESS 2008 Proceedings*, pages 302–307. IEEE, July 2008.

Supporting Single-GPU Abstraction through Transparent Multi-GPU Execution for Real-Time Guarantees*

[Extended Abstract]

Wookhyun Han, Hoon Sung Chwa, Hwidong Bae, Hyosu Kim and Insik Shin
KAIST, South Korea
insik.shin@cs.kaist.ac.kr

Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Recently, multi-GPU platforms appear to be a promising platform that allows two or more GPUs to work in parallel to accelerate computation or to accommodate a demand exceeding a single GPU's capacity. However, little support is provided for GPGPU over multi-GPU. In this paper, we consider supporting real-time applications in multi-GPU systems.

Despite many benefits of GPGPU, it raises several challenges to apply GPGPU technology to real-time computing. In the current GPGPU programming frameworks, applications (i) copy input data to the device memory from the host memory, (ii) launch a piece of GPU-accelerated code, called a *kernel*, on GPU to perform computation for outputs, and (iii) copy the device outputs to the host memory. Due to the non-preemptive nature of copying data to/from the device memory and launching kernels on GPU, it could block other data copy and kernel launch requests by higher-priority applications. To overcome such a non-preemptive nature a couple of studies [1, 3] share the principle of making non-preemptible regions smaller to allow higher-priority application to experience shorter blocking times for kernel launch [1] and for data transfer between host and device memory [3].

Supporting real-time computing on multiple resources (i.e., multiprocessors) is generally much more complicated compared with the single resource case. Liu and Layland [5] attributed the cause of difficulty in global multi-resource real-time scheduling to "*the simple fact that a task can use only one processor even when several processors are free at the same time.*" Such a task-level single-resource restriction causes optimal preemptive uniprocessor scheduling algorithms, RM and EDF, to become subject to a scheduling anomaly, called *Dhall's effect* [2]: some task sets may be unschedulable on multiple processors even though they have a low utilization close to 1. This shows the importance of relaxing the task-level single-resource restriction when possible.

Aiming at exploiting the massive parallelism of GPU architecture, each single kernel is typically designed to spawn a large number of threads that perform the same computation over different data in parallel. The nature of such a thread parallelism within a kernel enables the kernel to be decomposed into multiple sub-kernels, where each sub-kernel has a smaller number of threads, and individual sub-kernels to execute concurrently without interfering each other on different GPUs [4]. However, such a feature is not yet supported by the current prevailing GPGPU programming models and runtime supports such as CUDA and OpenCL.

Proposed Research.

We are currently exploring the benefits and issues of relaxing the restriction of executing a single kernel only on a single GPU in real-time multi-GPU systems. Composition of multiple GPU resources provides more capacity to be utilized by GPGPU applications, as well as more potential for performance improvement through multi-GPU parallel execution of each application. The multi-GPU execution of a single kernel can reduce GPU computation time leveraging the concurrent use of multiple GPUs. However, it imposes some overheads, including extra data transfer between host and device memory. Thereby, multi-GPU execution might not be beneficial to all GPGPU applications, but some depending on their compute- or memory-intensive behavior. This raises an issue of determining the multi-GPU mode of individual applications, i.e., determining how many GPUs each application uses. This entails a good strategy of making decisions from the system's perspective to optimize the system schedulability.

In addition, designing a good scheduling algorithm is also important. We aim to apply the virtual cluster-based scheduling approach presented in [6] for scheduling GPGPU applications with the multi-GPU mode. Depending on the GPU execution mode, each application is assigned to a GPU cluster, applications in each cluster are scheduled among themselves, and clusters in turn are scheduled on a multi-GPU system. There are a lot of new issues arise to adopt cluster-based scheduling in multi-GPU systems. One of them is how to define a cluster interface that takes multi-GPU execution overhead into account.

1. ACKNOWLEDGMENTS

This work was supported in part by BSRP (NRF-2010-0006650, NRF-2012R1A1A1014930), NCRC (2012-0000980), IITP (2011-10041313, 14-824-09-013) and KIAT (M002300089) funded by the Korea Government (MEST/MSIP/MOTIE).

2. REFERENCES

[1] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in gpgpus. In *ECRTS*, 2012.
[2] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.
[3] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *RTSS*, 2011.
[4] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *PPoPP*, 2011.
[5] C. Liu and J. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
[6] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.