Journal of Theoretical and Applied Information Technology
30th September 2017. Vol.95. No.18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

PLATFORM MODEL COMPOSITION FRAMEWORK FOR THE DEVELOPMENT OF REAL-TIME CONTROL SYSTEMS

[1]SANGSOO PARK
[1] Dept. of Computer Science & Engineering, Ewha Womans University, Seoul 03760, South Korea
E-mail: [1]sangsoo.park@ewha.ac.kr

ABSTRACT

Recent trend imposes stringent requirements on the design of embedded systems, differentiating them from general-purpose computer systems such as power consumption, timeliness, reliability, etc. Traditional platform models based on a single view of the underlying support software and hardware are not adequate for modeling all of the interference between the run-time tasks, resources, and services within a system caused by their behaviors and properties. This is mainly due to the fact that application software is tightly tuned to a particular platform, or it is designed and developed to be platform-specific. To overcome this limitation, a platform modeling framework for model-based software development is proposed by identifying the ranges of acceptable platform properties for application software and by specifying the models of computation with respect to the nonfunctional constraints on the underlying execution platform. Specifically, the focus is a multicore-based compositional platform model with fault tolerance for the developed framework. As a case study, a multicore real-time scheduling algorithm is applied to the framework, and the simulation results demonstrate the efficacy of the usability of the framework for supporting fault tolerance. Our proposed approach outperforms by 8.5% even for very heavy loaded system compared to the existing method.

Keywords: *Platform model, Model composition, Multicore, Model-based development, Embedded control system, Nonfunctional property*

1. INTRODUCTION

Computing and networking have become an essential part of everyone's daily life and business and have been deeply embedded in personal electronic devices such as cell phones and digital cameras as well as industry and military equipment such as robots, autonomous vehicles, and airplanes. Diverse modern devices and equipment, however, demand far more capabilities and functions from embedded systems than their predecessors. This trend imposes stringent requirements on the design of embedded systems, differentiating them from general-purpose computer systems. In particular, embedded systems must meet many "nonfunctional" constraints such as timing and resource (e.g., power consumption and memory capacity) constraints, a low manufacturing cost, and a short time-to-market.

Application software designers often overlook the effects of the underlying support software (i.e., operating system and/or middleware) as well as hardware such as on-board processors and communication devices, which are collectively called an execution platform or a platform. A platform typically has capabilities for data processing and storage and communication with external processors and physical devices such as sensors and actuators. On the other hand, application software is usually mapped to a set of run-time tasks, corresponding to threads or processes, which are well-known abstractions of concurrently executing applications in operating-system design [1]. A platform is therefore capable of simultaneously executing one or more application tasks. At the same time, the individual "system" services provided by the platform may be used by multiple application tasks.

The hardware in an embedded system also has obvious effects on the system's performance. As the underlying support software intervenes between hardware and application software, the functionalities and performance of the system software also have significant effects on the system performance. For application software in general-purpose computer systems, changes in the platform, especially with faster or slower hardware, may affect the "application responsiveness" without modifying the application software. However, for applications that are critical or sensitive to time

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

(real-time constraints) or resources (memory size or power consumption), the entire system must often be redesigned or recalibrated to meet such nonfunctional constraints. For this reason, safety-critical systems such as automotive and flight controls often have to use outdated processor technologies that had been verified or certified for the operating environments in which the applications execute.

This is mainly due to the fact that application software is tightly tuned to a particular platform, or it is designed and developed to be platform-specific. Such application software will require extensive rewrites of platform-dependent code, recalibration, or tuning parameters when the platform is changed; this is very expensive in both time and cost. Moreover, platform-specific software can also increase the design complexity, as the application software tends to depend on a vast majority of platform-specific services or technologies. Ideally, platform-independent software design, in which the irrelevant details of the underlying platform can be disregarded, would enable simpler and more portable application software [2]. However, this is difficult to achieve in real-time embedded systems with today's rapidly changing technologies.

In order to enable software modularity to avoid extensive rewrites of platform-dependent code, there have been many research efforts to define a standard modular software infrastructure for application software. For example, the AUTOSAR partnership, an alliance of automotive manufacturers and suppliers, has recognized the problem of software integration as a major challenge such that a number of open industry standards for automotive software frameworks have been developed [3]. However, these efforts are limited to the modularity of software components among different software developers/vendors and have not addressed how to deal with nonfunctional constraints such as timing and resource constraints.

Other than the modularity, the dynamic interactions among software components that result from various nonfunctional dependencies due to task scheduling, IPC, I/Os, and so on must be defined and modeled. The concurrent interactions among software components complicate the analysis of nonfunctional properties, and there is no obvious solution for these problems because platforms are often shared by multiple applications that contend for the same resources (e.g., CPU and memory), thereby interfering with each other. It is very difficult to predict the effects of this interference for all cases that might occur.

In this paper, to remedy or alleviate the above problem, a platform that can be analyzed in terms of the nonfunctional properties and a compositional model for the platform that exploits the dynamic and concurrent interactions are proposed. The usability of the developed platform model will be significantly enhanced by minimizing/eliminating the need for calibrating and/or redesigning application software when embedded systems are developed and/or transferred to a new platform. This will, in turn, reduce the time and cost for developing embedded application software.

In this study, the ranges of acceptable platform properties for application software are identified, and models for computation with respect to nonfunctional constraints are specified on the basis of a multicore platform model. The relationship between the application software and the execution platform is examined to develop a method for analyzing the nonfunctional properties more accurately to account for the platform effects on the design and execution of application software. To achieve this, application software and its nonfunctional requirements will be used to build a software model, which will then be automatically integrated with the constructed platform model by capturing the run-time properties of an entire system. Finally, the design and prototyping of a platform model are proposed, and its usability and efficacy are demonstrated with a case study, which applies the platform model to support fault tolerance in the system.

This paper is organized as follows. Section 2 describes the main components of the proposed framework for application-specific platform model generation and their integration based on model-based software development [4]. Section 3 presents a case study to demonstrate the usability efficacy of the developed platform model prototype to support fault tolerance. The paper concludes with Section 4.

2. PLATFORM MODEL FRAMEWORK

The proposed platform model is intended to provide software designers with a systematic way of dealing with design and run-time complexities resulting from both hardware and system software for real-time embedded applications. Current and future embedded system designs require the involvement of multiple stages, multiple groups of people, multiple disciplines, and multiple aspects [5]. Each may add complexity to the system. The representation of a design while limiting the

Journal of Theoretical and Applied Information Technology
30th September 2017. Vol.95. No.18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

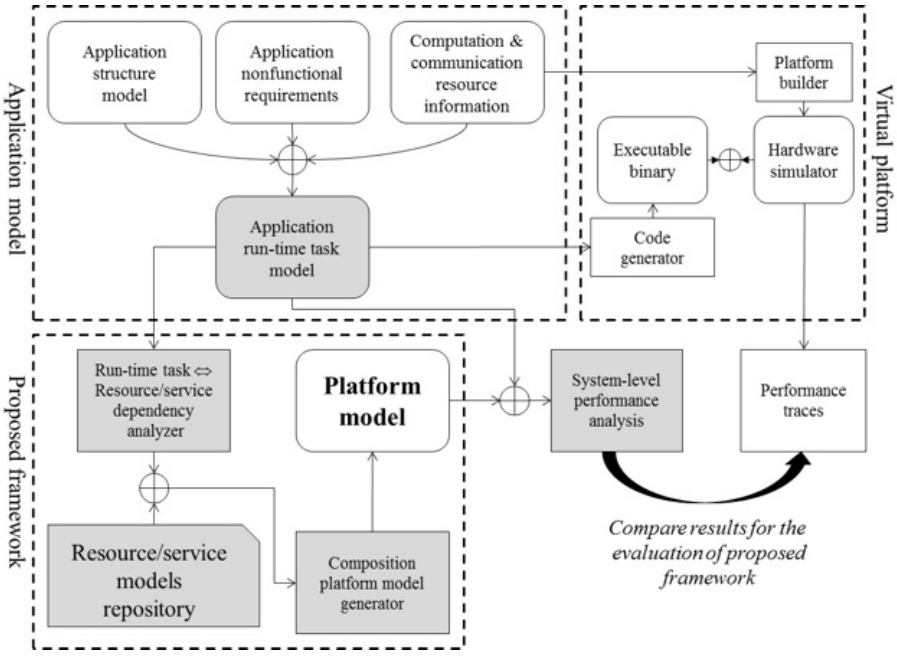


Figure 1: Framework for Automatic Platform Model Generation

information exposed to a group, which focuses on only a single discipline and system aspect, and sharing a design among different groups and stages to allow collaboration are great challenges. Further, methods of identifying and assessing the information that is needed at which stage, regarding which aspect, and for which group are necessary so that a lean process can be maintained and unnecessary complexity can be avoided. For example, a common set of APIs for the services provided by a platform with a resource budget may be provided to software component developers.

To ensure the satisfaction of nonfunctional requirements, however, a system-level analysis of the run-time properties, which are difficult to obtain or predict at the time of design, must be performed. Traditionally, nonfunctional constraints are handled by addressing them one-by-one via code-level optimization. However, current model-based software development often requires a high-level decision on design alternatives during the process of integrating the software model with the platform model [6]. The size and diversity of embedded systems are growing to meet the demand for an increasing number of functions, which in turn introduces design complexity. The run-time complexity mainly originates from dynamic environments and service expectations. This paper focuses on the development and integration of a platform model that provides fault tolerance at run-time with a low overhead for multicore-based real-time embedded systems.

2.1 Automatic Platform Model Generation

The platform in an embedded system is shared by multiple applications and tasks, frequently contending at run-time for the resources and services that the platform provides. This makes a single instance of the platform model reflect only a single view of its behaviors and properties, as all of the run-time tasks, resources, and services interfere with each other's behaviors and properties. To obtain a comprehensive view of the platform, a platform may be modeled "manually" by dealing with all types of interference one by one, but it is not practical or possible in many cases. There should be a way to generate a platform model automatically, which analyzes a given set of applications and then combines resource and/or service models that contain their independent behaviors and nonfunctional properties that the tasks make use of at run-time so that the constructed platform model models the possible interference that might occur simultaneously. Therefore, an "automatic platform model generation framework," which will be used as an appropriate representation for the functional and nonfunctional models of a platform, is developed in this study. This representation should be sufficiently expressive to contain resources and services together with their dependencies and interactions.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Although the framework itself will be domain-specific, the methods used to construct such a framework should be generic for any type of system.

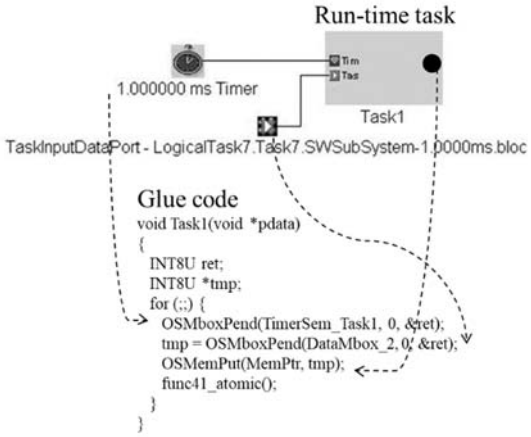


Figure 2: An Example of Glue-Code Generation for the uC/OS Real-Time Kernel

As depicted in Figure 1, the framework is composed of several components: (i) the generation of an application run-time task model, (ii) an analysis of run-time tasks and platform dependencies, (iii) a repository of the behaviors and nonfunctional properties of the resource/service models, (iv) the generation of a platform model by composing resource/service models from the repository with the dependency analysis results, and (v) a system-level analysis of the integrated application run-time tasks and platform model. These components are essential for the model-based software development process to proceed from an early design stage to a sequence of refinements of its design.

To support the run-time analysis, the application structure with nonfunctional requirements is refined to a set of run-time tasks, which correspond to a process or thread in general operating systems. The run-time properties are automatically assigned to each task on the basis of a given platform model in order to avoid violating the nonfunctional constraints. Note that the constructed run-time tasks must confirm the concurrency or serialization of their execution.

It is possible to manually integrate models according to each run-time task. However, it is highly desirable to analyze the dependencies between run-time tasks and resources/services for the model integrations automatically for fast and error-prone processes. For this, the novel method shown in Figure 2, which was presented in a previous work, is used to automatically generate glue code, which fits into the APIs that the underlying support software provides, by analyzing the run-time tasks and integrating them into the platform [5]. This method can be converted for the proposed framework to analyze the dependencies from the run-time tasks to the platform. This is considered a key benefit since the automatically built models will significantly help designers to obtain the complex dependencies and interactions in the system, and it will enable a significantly fast and accurate system-level run-time analysis.

2.2 Composition of the Application Run-Time Model and Generated Platform Model

To design and develop a real-time embedded system while meeting all nonfunctional requirements, the design and/or run-time properties of both the "application model" and "platform model" where the applications will run must be captured. A system-level analysis can then be performed to verify and ensure the satisfaction of the nonfunctional requirements. To this end, a compositional model [7] must be generated with the results of a dependency analysis using the resources and/or services provided by the underlying platform for each application task. As shown in Figure 3, a separate model for the computations and nonfunctional properties can be generated to schedule a task on one CPU in the proposed platform model framework for each entity, such as the timer interrupt and bus contention.

In this approach, an output of a resource or service entity is turned into an input of another compositional model. The system-level run-time analysis can be obtained with the execution flow of the models that are solved iteratively. To do so, sub-model integration of the multiple inputs and the propagation of the intermediate models to the subsequent models along with the execution flow must be carefully defined by the high-level abstraction.

Moreover, the system considered in this proposed framework consists of N tasks running on an M-core system. All cores are assumed to have identical CPUs with synchronized clocks. Each core behaves as a single processor and can run at most one task at any given time. To reduce the analysis complexity, it is assumed that the execution of tasks is synchronized with the periodic timer ticks so that the CPU time is allocated in a time slot, which will be referred as the time quantum t representing the time interval [t, t + 1] [8].

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

For example, the compositional models for a run-time task, $Task_1$, in Figure 3 can be defined by logic operations, i.e., the AND, OR, and NOR operations. The inputs to $Task_1$ include one synchronous event by a timer and asynchronous data by another task.

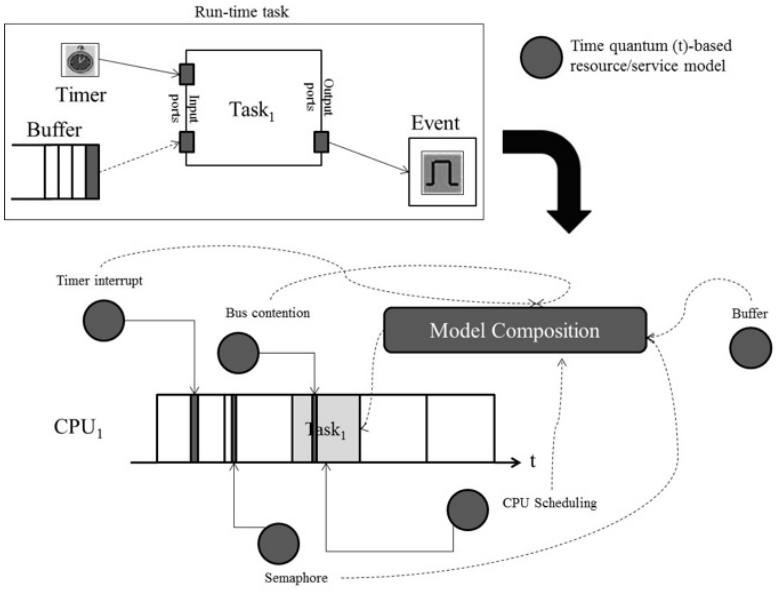


Figure 3: An Example of Model Composition for Scheduling $Task_1$ on CPU_1

Thus, it must pend until a timer event is active, though it may proceed regardless of data availability in the buffer. The timer event is the output of a semaphore model driven by timer interrupts, whereas the asynchronous data solely depend on a memory buffer model. Then, as soon as a CPU scheduling model grants a time quantum t, $Task_1$ starts its execution. However, the execution of $Task_1$ may be suspended by the event of bus contentions or any interrupts to CPU_1, where the time quantum is assigned.

3. CASE STUDY: FAULT-TOLERANCE SUPPORT

For mission-critical applications such as flight automotive controls, fault-tolerance is the most important nonfunctional property in the system, although it may be less critical to other applications, such as smartphones and consumer electronics. To provide high reliability in mission-critical real-time control systems, each realized on a multi-core SoC, the functional or nonfunctional constraints should be met even in the event of transient or permanent failures.

To show the effectiveness and applicability of the proposed approach in this paper, a case study to fulfill the fault-tolerance of a system using the characteristics of multi-core SoC. That is a task can be duplicated before an occurrence of a fault or migrated in the event of a fault and then be switched to ready core immediately with low overhead. This type of task migration or duplication may not be feasible on a network of multiple controllers with single CPUs due to the long delay in migrating tasks between separate CPUs, and also communicating between controllers, as opposed to within a single chip.

This enables scheduling multiple versions of each task on different cores to provide fault-tolerance [9, 10, 11]. For periodic real-time tasks, static allocation and scheduling can be used [9], where tasks are allocated and scheduled on different cores prior to their execution.

3.1 Software Component and Task Duplications on Idle Cores for Fault Detection

In this paper, duplication approach on multi-core SoC at task level is applied to provide fault-tolerance. The system model assumes a Real-time Operating System (RTOS) is running on a multi-core configured as a symmetric multiprocessor and core failures do not occur during the execution of the fault-tolerant method, because the execution of the proposed methods will take much less time than the the MTBF (Mean Time Between Failures) of a core [12].

In the proposed method, one global scheduler exists where all jobs of tasks arrive and each core

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

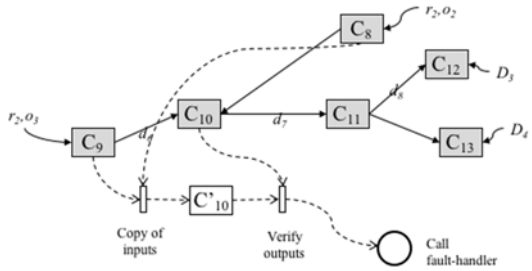
receives tasks from the global scheduler. When a job of task is released, the global scheduler determines whether to create a redundant task to continue to run in case of task failure on the original task. Once the redundant task is created the scheduler determines how to allocate the original task with a redundant task onto available cores. Note that different versions of the same task cannot be allocated to the same core to provide fault-tolerance. Also, once a task is allocated on a core then it must execute entirely on the core and its context is maintained on that core. However, different jobs of the same task may execute on different cores.

To provide fault-detection methods in the platform model framework, two duplication methods are applied at the time of design and run-time. In the application structure, any software components can be duplicated—most likely, the ones that are more dependent (or have more inputs) so that the selection would increase the fault-detection coverage or those that are as early as possible in order to minimize the fault-detection time—at the time of design, as long as the duplications do not violate the resource constraints. As shown in Figure 4(a), the software component C_{10} can be duplicated as C'_{10}. However, it should be noted that all inputs must be duplicated as well in the context of C'_{10} in order to maintain the data integrity in the original software component. Note that Table 1 summarized notations used in Figure 4.

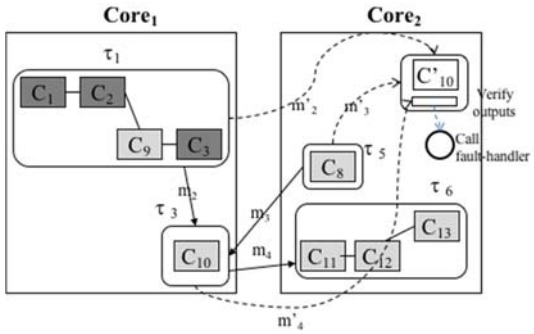
Table 1: Notations

C_i	Software component
r_i	Invocation rate
o_i	Release offset
D_i	Deadline
d_i	Data
$Core_i$	CPU core number
τi	Task
m_i	Message
$\rightarrow$	Synchronous input (precedence relation)
$\cdots\rightarrow$	Asynchronous input (data / message)

Then, the outputs of C_{10} and C'_{10} would be compared afterwards so that the platform detects any faults. This approach has the advantage that the timing and resource properties may be guaranteed at the time of design time; however, all procedures must be statically assigned in the platform. Thus, some resources must be reserved for fault detection.



(a) Software Run-Time Model



(b) Task Model

Figure 4: An Example of Fault Detection in the Proposed Platform Model

In contrast, any tasks can be duplicated at run-time, as shown in Figure 4(b). A CPU scheduler may detect an idle cycle on some of cores; then, it would fork an instance of a task for duplication to detect faults. Though the context of a duplicated task is independent of the original one, the inputs to the task delivered by external messages or IPCs must be re-sent or multi-casted at run-time. This approach has the advantage of fully utilizing the idle time on multicore processors; however, the support software in the platform must provide features such as detecting idle cycles and duplicating inputs.

However, models and algorithms should be developed to choose the component or task that is about to be duplicated to minimize fault-detection latencies and/or maximize the detection coverage of a fault. In this study, the primitives are devised and evaluated which are required to detect the idle time

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

quantum in the proposed platform model and to duplicate the software component and run-time task, including their inputs and the basic logic to compare the outputs that determine any occurrence of faults.

utilization compared to the partitioned scheduler [14]. Note that throughout the experiments with the simulator and implementation, the time quantum of the scheduler was set to 1 *ms*.

An extensive set of experiments is performed based on randomly generated tasks. In each

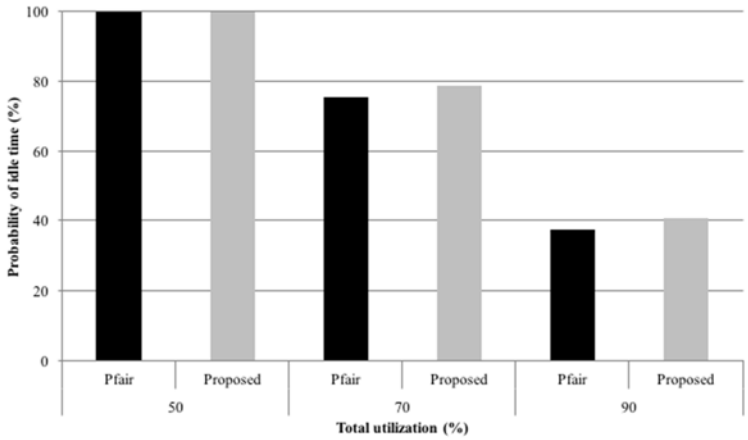


Figure 5: Probability that a Time Quantum is Idle when U=50%, 70%, and 90%

3.2 Implementations and Simulation Results

A discrete-time simulator is built to evaluate the applicability and effectiveness of the proposed platform model framework in terms of the fault tolerance. The resource/service model for CPU scheduling described in Section 2 is implemented on top of the simulator. In addition to the simulator, the proposed platform is ported to an embedded multicore chip that is configured with four cores, operates at 100 MHz, and is simulated by the RealView system-level hardware system simulator [13]. This simulator is known to require only a moderate amount of simulation time, thus making it suitable for embedded real-time multicore systems. The simulator can mimic various multicore CPUs and model different types of memory and a range of cache architectures and external peripherals that can be customized.

In this paper, two different simulations are performed to show the effectiveness and applicability of the proposed framework. The one is to estimate the proportion of idle cycles that can be used for fault detection and the other is to estimate number of necessary cores to guarantee all task timing deadlines in the case of one core failure.

To measure the core CPU scheduling algorithm, Pfair was used as the baseline for comparison. The Pfair scheduler allows tasks to migrate from one core to another within the chip, achieving full

experiment, tasks were generated with randomly chosen parameters that are representative of those in an automotive control application. In these experiments, task sets are randomly generated using the GNU Scientific Library (GSL) [15], with the total utilization ranging from 2.0 to 4.0 with a step size of 0.1. Note that the base utilization bound of 50% (i.e., U = 2.0, where M is the number of cores on the chip [14]) was chosen on the basis of a real application—the electronic throttle control (ETC) in the powertrain control system presented in [16].

The experimental results for the proportion of idle cycles that can be used for fault detection are presented to show the applicability of the proposed framework. To detect software/hardware faults, an instance of a software component or run-time task is duplicated upon detecting idle cycles by the CPU scheduler. The outputs generated by the original software component or run-time task are then compared with the outputs of the duplicated one, which is executed on idle cores [8]. The performance, e.g., the latency or coverage, of this fault-detection service highly depends on how the idle time quanta are distributed over time. Therefore, the probability that an idle time quantum exists with respect to the total utilization bound of the randomly generated task sets on a four-core processor is measured

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

As shown in Figure 5, the probability that a time quantum is idle is 100% when the task utilization is 50%. Full duplicate copy of a system can be executed in this case, similar to the primary and backup approach. However, our proposed platform still has idle cores with a probability of 40% for very heavy loaded system, i.e., when U = 70% and 90%. The proposed platform always outperforms Pfair.

On the other hand, since multiple versions of each task are created, stored separately in memory and run in parallel, meaningful amount of computing and storage resources can then be inevitably increased in terms of financial and power costs. Therefore, the number of necessary cores to guarantee all task timing deadline in the case of one core failure is also measured to estimate the overhead due to the support of fault-tolerance.

For this, a heuristic suboptimal task allocation algorithm when one core failure occurs and the corresponding scheduling algorithm to account for task duplications is developed. In the experiments the period of each task are uniformly distributed between 5ms and 100ms. The maximum task utilization within a given task set is set to be 0.2 and 0.5.

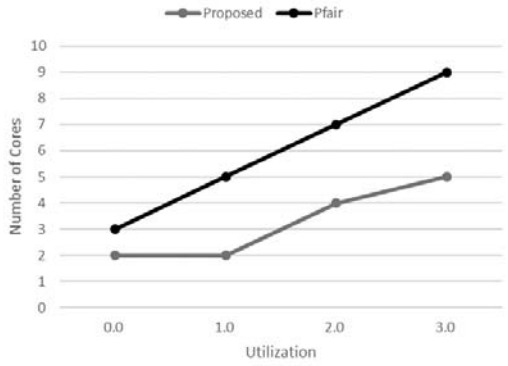


Figure 6: The Number of Necessary Cores in case of One Core Failure when U=20%

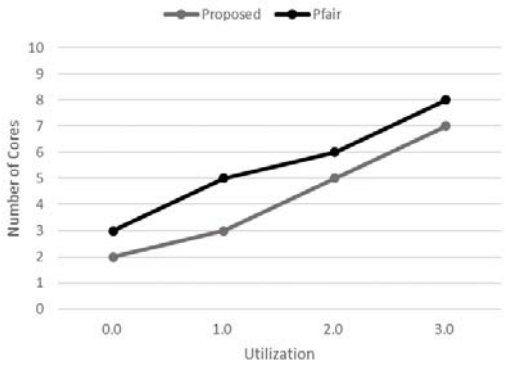


Figure 7: The Number of Necessary Cores in case of One Core Failure when U=50%

As shown in Figure 6 and 7, the number of cores required to meet all the timing deadline even though one core is failed is always less than the case if Pfair is used.

The experimental results show that although the proposed approach inevitably requires additional redundant resources such as CPU cores, memory space, and so on, however, it efficiently provides fault-tolerance compared to the existing approaches.

4. CONCLUSIONS

This paper presented the development, implementation, and evaluation of an automatic platform model generation framework that examines the relationship between the application software and the execution platform and deploys a method for the accurate analysis of the nonfunctional properties to account for the platform effects on the design and execution of application software. To achieve this, the application software and its nonfunctional requirements are used to build a software model, which will then be automatically integrated with the constructed platform model by capturing the run-time properties of an entire system.

The proposed platform model is intended to provide software designers with a systematic way of dealing with design and run-time complexities resulting from both hardware and system software for real-time embedded applications. The key components of the platform model framework are the generation of an application run-time task model, the analysis of run-time tasks and platform dependencies, the building of a repository for the behaviors and nonfunctional properties of the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

resource/service models, the integration of a platform model by composing resource/service models from the repository with the dependency analysis results, and the deployment of a system-level analysis of the integrated application run-time tasks and platform model. A resource service model that supports fault tolerance is deployed to evaluate the proposed framework, and the simulation results demonstrate its usability and efficacy in supporting fault tolerance in a multicore real-time control system. Our proposed approach outperforms for very heavy loaded system compared to the existing approach, e.g. by 4.4% when the CPU utilization is 70% and 8.5% when the CPU utilization is 90%.

In the proposed approach, since multiple versions of each task are created, stored separately in memory and run in parallel, meaningful amount of computing and storage resources can then be inevitably increased in terms of financial and power costs. However, the experimental results shows that multicore scheduling algorithm can be applied to support fault-tolerance with higher probability to detect a fault and to continue running an assign task and with smaller number of cores to tolerate the failure compare to the previous work.

While this paper has demonstrated the usability of the thus-developed platform model, many opportunities for extending the scope of this paper remain such as minimizing the need for calibrating nonfunctional properties of application software as well as underlying hardware and system software when embedded systems are developed and transferred to a new platform.

ACKNOWLEDGMENTS:

This article is an extension of the following paper: Sangsoo Park, "A Platform Model Framework for the Development of Real-Time Control Systems", *International Conference on Computing Convergence and Applications (ICCCA'16)*, Busan, 2016, pp.109-112. This work was supported by the National Research Foundation of Korea funded by the Korean Government (NRF-2017R1D1A1B03030393).

REFRENCES:

[1] Park, S., Olds, W., Shin, K. G., and Wang, S., "Integrating Virtual Execution Platform for Accurate Analysis in Distributed Real-Time Control System Development", *International Real-Time Systems Symposium (RTSS '07)*, IEEE, Tucson, 2007, pp. 61-72.

[2] Selic, B., "Accounting for Platform Effects in the Design of Real-Time Software Using Model-Based Methods", *IBM Systems Journal*, Vol. 47, No. 2, 2008, pp. 309-320.

[3] Fürst, S. and Bechter, M., "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform", *International Conference on Dependable Systems and Networks Workshop (DSN '16)*, IEEE/IFIP, Toulouse, 2016, pp. 215-217.

[4] Goswami, D., Lukasiewycz, M., Steinhorst, S., Masrur, A., Chakraborty S., and Ramesh, S., "Model-Based Development and Verification of Control Software for Electric Vehicles", *Proc. of Design Automation Conference (DAC '13)*, ACM, Austin, 2013, pp. 1-9.

[5] Park, S., Shin, K. G., and Wang, S., "Integration of Collaborative Analyses for Development of Embedded Control Software", *Proceedings of the IEEE*, Vol. 98, No. 4, 2010, pp. 546-461.

[6] Zaki, M. Z. M., Jawawi, D. N. A., and Isa, M. A., "Integrated MARTE-Based Model for Designing Component-Based Embedded Real-Time Software", *International Journal of Software Engineering and Its Applications*, Vol. 9, No. 3, 2015, pp. 154-174.

[7] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., and Ernst, R., "System Level Performance Analysis – the SymTA/S Approach.", *IEE Proceedings Computers and Digital Techniques*, Vol. 152, No. 2, 2005, pp. 148-166.

[8] Park, S., "Robust Scheduling of Dynamic Real-Time Tasks with Low Overhead for Multi-Core Systems", *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*, Vietri sul Mare, 2013, pp. 69-76.

[9] Beitollahi, H., and Deconinck, G, "Fault-Tolerant Partitioning Scheduling Algorithms in Real-Time Multiprocessor Systems", *The 12th Pacific Rim International Symposium on Dependable Computing*, IEEE, Washington, 2006, pp. 296–304.

[10] Ghosh, S., and Melhem, R., "Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System", *International Parallel Processing Symposium*, IEEE, Cancun, 1994, pp. 775–782.

[11] Krishna, C. M., and Shin, K. G., "On Scheduling Tasks with a Quick Recovery from Failure", *IEEE Transactions on Computers*, Vol. 35, No 5, 1986, pp. 448-455.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[12] Srinivasan, J, Adve, S. V., Bose, P., and Rivers, J. A., "The Impact of Technology Scaling on Lifetime Reliability", *International Conference on Dependable Systems and Networks (DSN'04)*, IEEE, Florence, 2004, pp. 177-187.

[13] ARM, *Design Simulation Model User Guide* http://infocenter.arm.com/help/topic/com.arm.doc.dui0302d/index.html, 2015.

[14] Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, A., and Baruah, S., "A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms", in the *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Chapman and Hall/CRC, 2004.

[15] GNU, *GNU Scientific Library*, http://www.gnu.org/software/gsl/.

[16] Ishikawa, M., McCune, D. J., and Saikalis, G., "CPU Model-Based Hardware/Software Co-Design for Real-Time Embedded Control Systems", *Proc. of the SAE World Congress (SAE '07)*, Detroit, 2007, p. 2007–01–0776.