Integrating Virtual Execution Platform for Accurate Analysis in Distributed Real-Time Control System Development

Sangsoo Park, Walter Olds, and Kang G. Shin
EECS Department, The University of Michigan
{ssoopark,wolds,kgshin}@eecs.umich.edu

Shige Wang
General Motors R&D and Planning
shige.wang@gm.com

Abstract

A distributed real-time control system is modeled by automatically generating a virtual execution platform and integrating it with an abstract run-time model. This allows us to capture the dynamic effects of non-deterministic behavior of the underlying hardware and real-time operating system (RTOS), which cannot be accurately evaluated by existing static approaches. Our framework has been implemented and integrated with the existing AIRES toolkit. The construction of a virtual execution platform for a control application is observed to take 422ms. The integrated platform and control application consists of 56 software components connected by 1280 links for data exchanges and precedence relations, and the constructed platform executes the application at a normalized speed—defined as the ratio of simulation time to real time — 10.6. Our preliminary evaluation has demonstrated the virtual execution platform's capability of providing accurate run-time information at a reasonable time-cost.

1. Introduction

Embedded software for large-scale distributed real-time systems, such as automotive control or avionics systems, is complex and must deal with many interacting control functions. The correctness of such systems depends not only on the results of functional computation, but also on 'non-functional' properties, such as response time and resource consumption. The need for accounting for these properties complicates the development of embedded software.

To develop real-time embedded software that meets the non-functional requirements, it is essential to characterize the available software components in terms of their required properties, such as timing and resource consumption. It becomes much more difficult to meet system-level requirements when the components of a large-scale real-time embedded system must be (i) distributed over multiple processors connected by a communication network, and (ii) integrated to meet both the functional and non-functional requirements.

In our previous work on AIRES (Automatic Integration of Reusable Embedded Software) [1], we developed and implemented a suite of methods and tools for this purpose. As shown in Fig. 1, the application structure and its non-functional requirements are captured in a software model, and the computation and communication resource information is captured in a platform model. These models are then combined to construct an application run-time model, which expresses the run-time properties of the application.

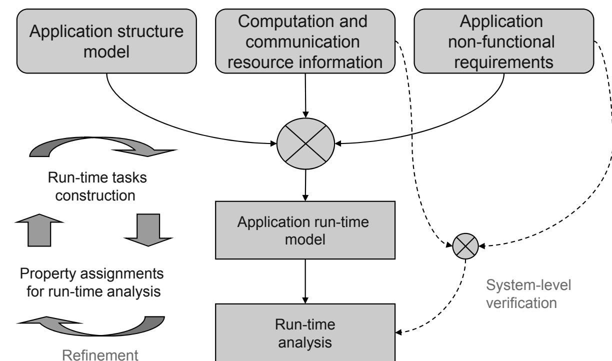


Figure 1: Design and verification of distributed real-time systems with the AIRES toolkit

An important issue that has not been addressed before is the run-time properties of the execution platform that includes the underlying support software (i.e., RTOS) and hardware (on-board processors and communication devices). Although the application run-time model in Fig. 1 contains all of the static information needed for a run-time analysis at design-time, the non-deterministic behavior caused by the execution platform cannot be captured well in the run-time model. To ensure the non-functional requirements to be met, however, a system-level run-time analysis must include this non-deterministic behavior as well as the usually-assumed/available static information.

The traditional method for meeting the non-functional

1052-8725/07 $25.00 © 2007 IEEE
DOI 10.1109/RTSS.2007.21

IEEE computer society

constraints has been to address them, one-by-one, with code-level optimization. However, the need to meet multiple constraints in a large-scale distributed real-time control system becomes a high-level decision problem (on design alternatives) during the software component composition process. It is impractical to mock up a complete system at an early design stage, and capture more accurate run-time behaviors in order to verify the system-level requirements for each design alternative. A more practical (hence desirable) approach is to automatically generate a virtual execution platform and then integrate it with a given application run-time model that runs on a system-level simulator to obtain accurate run-time performance and behavior.

In this paper, we present a novel method for accurate and efficient generation of a virtual execution platform which can be used for design of distributed real-time embedded systems. This will enable high-level design decisions to be made on the basis of the results of integrating an application run-time model with its execution platform. We have developed a framework for generating a virtual execution platform and running the application on that platform to collect accurate run-time performance data. This framework has been implemented in, and integrated with, the existing AIRES toolkit. Our preliminary evaluation results on two control system applications have shown that the framework can provide accurate run-time information at a reasonable time-cost.

The rest of this paper is organized as follows. Section 2 presents the system model and assumptions to be used. Section 3 describes the architecture of an execution platform and its integration with the application run-time model. Section 4 presents the results of our preliminary evaluation based on a simple control system. Section 5 discusses the related work, and the paper concludes with Section 6.

2. System Model and Assumptions

We consider a typical distributed control system that consists of sensing (input), processing (computation), and actuation (output) components, as shown in Fig. 2. The information processing dependencies among these components impose precedence constraints between sensing and processing components, and between processing and actuation components. Moreover, the resource demands of these software components, such as total CPU time and network bandwidth, should not exceed the capacity of the execution platform.

2.1. Hardware

It is assumed that the computation and communication resources to be provided by the hardware—including the number of processors, a communication network, and the type of devices to be used—have already been determined during a previous design phase, by considering constraints, such as budget, space and weight limitations.

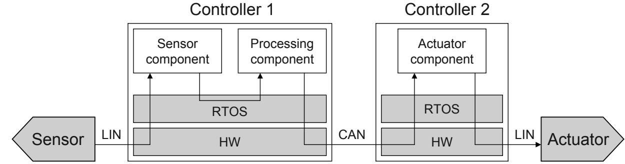


Figure 2: An example distributed automotive control system

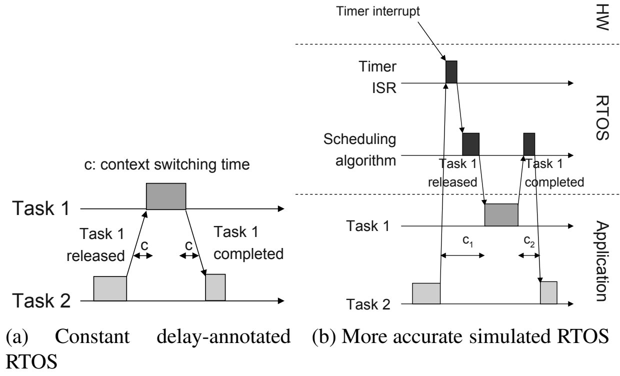
Each controller unit is configured as a set of hardware components to support an RTOS and external communications. Its minimum configuration includes an interrupt controller, one timer clock (typically generated by a crystal oscillator), and one or more communication devices, together with a CPU and memory. The interrupt controller handles external interrupts according to priority to cope with external events or data arriving from software components. A system-wide timer has the highest priority of all interrupt sources. All communication devices are assumed to have memory-mapped I/O and generate an interrupt when new data becomes available.

2.2. RTOS

An RTOS has often been considered as a set of services incurring a *constant* overhead manifested as a context-switch time or an inter-process communication delay, as shown in Fig. 3a. This is the usual way of modeling an RTOS in most embedded software design methods because it allows the use of existing real-time analysis techniques or formal methods for the verification of timing constraints [10].

This traditional constant-overhead RTOS model ignores interleaved, dependent executions of the RTOS and applications. Since it does not capture the dynamic behavior of an RTOS, it is difficult to obtain accurate performance data of the underlying execution platform for use in the run-time analysis. In our system model, however, RTOS service overheads are collected via accurate simulation based on the integration of the application and the RTOS. For example, the run-time behavior of CPU scheduling can be captured at a hardware level with an interrupt handler for the OS tick timer. We can also capture the dynamic behavior of the same functionality in the CPU scheduling, which may incur different overheads, such as c_1 and c_2 in Fig. 3b.

An RTOS provides a set of basic services, such as scheduling, inter-process communication, interrupt, and timing services. The scheduling policy is assumed to be



(a) Constant delay-annotated RTOS (b) More accurate simulated RTOS

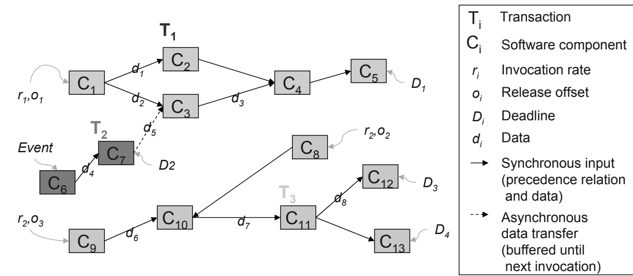
Figure 3: Comparison of RTOS run-time behaviors

priority-based and preemptive, as commonly adopted in commercial off-the-shelf RTOSes. We also assume that no priority inversion occurs at run-time, implying an appropriate design of the application run-time model to prevent priority inversions, or the RTOS's support of dynamic priority assignment to handle the priority inversion problem [3]. The IPC (inter-process communication) service, implemented with such primitives as mailboxes and semaphores, is assumed to support both blocking and non-blocking communications. We also assume that the interrupt service supports nesting to handle prioritized external interrupts, and the timing service has fine-granularity.

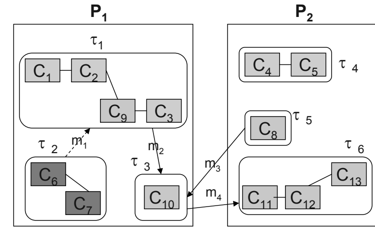
2.3. Applications

Each control process—a sequence of sensing, processing, and actuating—in a system is referred to as a *transaction* in our system model, and consists of software components, their interactions and non-functional requirements. The application software as a whole can then be represented as a set of concurrent transactions, as shown in Fig. 4a, and captured in a directed acyclic graph where each node represents a software component and each link represents a data exchange or precedence constraint between two components.

We assume that every component is associated with a worst-case execution time (WCET) as its computation resource demand, and the maximum memory usage as its memory resource demand for its code and the temporary data required during its execution. Each input component is specified by an invocation rate and release offset, and is triggered by a timer if a fixed invocation rate has been assigned, or by an event—which may be generated by the output of a component in another transaction or by data arriving at an external communication device—whereas each output component is specified by an end-to-end deadline. The input component and the subsequent components in a transaction are connected by links, either synchronous or asynchronous.



(a) Application structure with non-functional requirements



(b) Run-time tasks on two processors, P_1 and P_2

Figure 4: An example of an application run-time model

To support the run-time analysis shown in Fig. 1, the application structure with non-functional requirements is refined as a set of run-time tasks τ_i [19], corresponding to a thread or process, which is a well-known abstraction of concurrently executing applications. As shown in Fig. 4b, each software component is allocated to one (P_j) of the processors, and iteratively merged and ordered into tasks so that the sequence will run to completion (e.g., $C_1 \rightarrow C_2 \rightarrow C_9 \rightarrow C_3$ in τ_1): no task is allowed to contain any idle time, which may lead to unintended context switches by the RTOS. Run-time properties, such as release offset and priority, are then assigned to each task and the system-level end-to-end timing deadlines are similarly decomposed into a timing constraint for each task, while ensuring that the resource constraints are not violated.

During the construction of run-time tasks, the communication links between tasks are transformed to a set of synchronization or asynchronous data-transfer messages m_k. A message between two tasks corresponding to a synchronized communication between the components in these tasks carries a firing token, and the receiver task becomes ready to execute only after the arrival of the message. The execution of a task receiving only a message corresponding to an unsynchronized communication, on the other hand, depends on the scheduling decision made by the local supporting system, such as an operating system. Similar to the task formation, multiple messages can be merged to form a single message if they meet a set of defined criteria, such

as the same invocation rate and at most one of them with a firing token. The timing properties of messages, including their release offsets, invocation rates, and relative deadlines, are derived from those of their generator tasks. The resource consumption of a message is determined by the transmission delays of all its contained signals on a network link, which can be abstractly represented using the total size (bytes or bits of data) of all its signals.

3. Integration of Application Run-Time Model with Execution Platform

The system-level verification of non-functional requirements that depend solely on the static information captured by the application run-time model (described in Section 2.3) cannot describe the dynamic run-time behavior of an execution platform. The execution platform is only characterized by its computation and communication resources, expressed as the parameters of the analytical model which is used to confirm the system-level end-to-end timing constraints. Temporal uncertainties caused by the platform are, therefore, ignored.

Small glitches in a large-scale distributed real-time system can often lead to system failure if they breach timing or resource constraints, so these constraints must be thoroughly verified by accounting for both the static and the dynamic run-time behaviors of a system. As mentioned in Section 1, prototyping a complete system can address this problem, but it is inefficient in time and cost and is often infeasible at an early design phase. To remedy these problems, we have developed a framework that integrates the application run-time model with a virtual execution platform, as shown in Fig. 5.

3.1. Construction of a Virtual Execution Platform

We now show how an in-depth analysis of system performance with full-scale system simulation—that can run an application together with its RTOS—can be achieved before building a hardware prototype.

We use a cycle-accurate hardware simulator to construct a simulated hardware system for a controller unit described in Section 2.1. We generate executable software from an application run-time model with an RTOS on the simulated hardware system. This arrangement allows us to capture accurate run-time behavior and thus obtain accurate performance data on each individual controller unit. We then validate the system-level non-functional requirements. Although register transfer-level (RTL) and transaction-level simulators are commonly used as analysis tools in early stages of system design, they often turn out to be too slow [6]. We therefore use a cycle-accurate and highly-configurable system simulator (from the VaST systems [18]) to construct a virtual hardware system. This simulator is known to require only a moderate amount of simulation time, thus making it suitable for large-scale distributed real-time systems. Not only can the simulator mimic various CPU cores but it can also model different types of memory and a range of cache architectures and external peripherals that can be customized.

The simulated hardware system we constructed contains ARM926EJS and SH-2A CPU cores, and uses existing virtual hardware models for the CPU, memory and bus. However, we have built our own models of peripherals, such as interrupt controller, timer, and communication devices, so that we may customize their run-time behaviors and insert probes to obtain their timing and resource characteristics.

The run-time behavior and performance of an RTOS are highly dependent on its implementation and on the underlying hardware. Thus, it is inappropriate to use static analysis techniques when we need to account for accurate run-time behavior and performance data for the underlying execution platform. Therefore, instead of using an abstract RTOS model [7], we ported a real RTOS to the simulated hardware system and captured accurate run-time behavior and performance data. The limitation of this approach is that only a set of selected RTOSes can be used in the development process. However, RTOSes often provide common APIs (e.g., POSIX) so as to facilitate the development of multiple software components which have to be integrated within a large-scale embedded software system. Note that any RTOS that meets the requirements of our system model in Section 2.2 can be used in the integrated framework.

Here we have chosen a well-verified lightweight RTOS, uC/OS [12], which satisfies all of the above requirements. We have ported this RTOS to ARM926EJS and SH-2A cores by building board support packages (BSPs). We have also developed device drivers for peripherals—interrupt controller, timer, and communication devices—in the simulated hardware system.

3.2. Automatic Mapping of Run-Time Tasks to RTOS Glue Code

At this stage, we have a set of application run-time tasks with the non-functional requirements derived from the application run-time model, together with a virtual execution platform including the RTOS and hardware. Although the application run-time model possesses all the properties necessary for constructing run-time tasks which correspond to processes or threads in the RTOS, one-to-one mapping from a run-time property in the application run-time model to a corresponding RTOS service is not possible because that would involve the integration of two models of very different forms; one is an abstract model represented by a graph

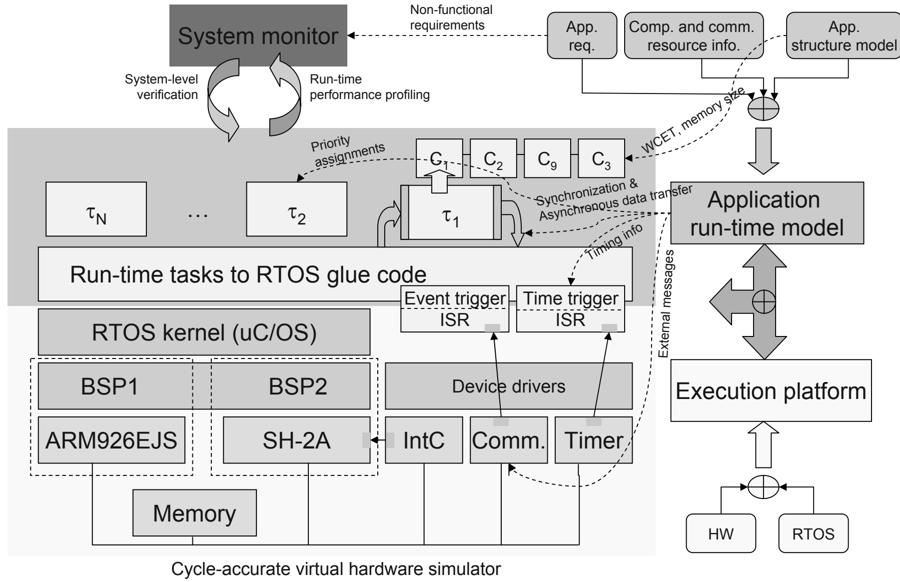


Figure 5: A framework for integrating the application run-time model and the execution platform

and the other already contains implementation details.

A programmer may write code to implement the abstract model for each run-time task, but it is highly desirable to automatically map the run-time tasks to the target RTOS glue code for software productivity and accuracy. To meet this need, we have developed a method for automatic generation of glue code that integrates the run-time model with the underlying execution platform at each of the levels shown in Fig. 5. Our method deals with software components and schedulable run-time tasks, and triggers input to components of the run-time tasks.

3.2.1. Software Components of a Run-Time Task

During the construction of run-time tasks, the process of sequencing components explicitly converts precedence constraints to a sequence of function calls to each component. This conversion is achieved by traversing the graph that represents an application run-time model, and by identifying the run-time task τ_i allocated to processor P_j that we are considering. The component sequencing for τ_i is implemented as a linked list, allowing us to call a function for any software component C_k. Fig. 6 shows an example task body—which we call a *task context*—of the run-time task τ_4 which consists of two components C_4 and C_5. There are precedence constraints between components, $C_4 \rightarrow C_5$, but there is no data transfer from C_4 to C_5. We will assume that data or messages from other run-time tasks are passed as an array of pointers `args` in Fig. 6. The data required to execute each component in τ_4 is assumed to be stored in local variables or in the stack.

```
void Task4Ctx (void **args)
{
    Comp4(args[0]);
    Comp5();
}
```

Figure 6: An example task context: run-time task τ_4

However, if data or messages need to be transferred within a run-time task, then extra memory space must be allocated to the context, because a component cannot directly access the data assigned to local variables in another component. We, therefore, declare a local variable for each data transfer, not in the component that requires the data, but in the task context, and the data is passed as an argument of the function call. The amount of memory required for τ_i is $\sum_{j \in \tau_i} sizeof(d_j)$, where j is a data connection within a run-time task i as shown in Fig. 4.

In an embedded system, the memory requirement must usually be kept as small as possible. During code generation, the memory requirement of a run-time task can be reduced by sharing space for local variables since each local variable is only required during a single data transfer. Table 1 shows how the minimum memory space requirement is calculated for each run-time task. In this example, the task consists of four components with the sequence $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$. The arrows in the figure represent precedence constraints, while $d_1 : 3$ is a data connection from C_1 to C_2 of size 3, and so on, for d_2 and d_3. The total memory requirement is $\max \sum_s sizeof(d_{i,s})$, where s is a

sequence number and $d_{i,s}$ is the corresponding active data. In this case, the space required is reduced from 9 to 7.

Table 1: An example of calculating minimum memory space within a run-time task

	C_1	C_2	C_3	C_4
$d_1 : 3$	$3 \rightarrow$	$3 \rightarrow$		
$d_2 : 4$	$4 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	
$d_3 : 2$			$2 \rightarrow$	$2 \rightarrow$
Total	7	7	6	2

3.2.2. Synchronization and Asynchronous Data Transfer between two Components

A run-time task is implemented as a process or a thread in the RTOS. To synchronize two tasks or to transfer data between two tasks, an OS trap is required by the primitives providing the corresponding RTOS service, as summarized in Table 2. We must ensure that all the run-time tasks and messages, except for asynchronous data transfers, are synchronized and that the order of execution follows the application structure and meets the non-functional requirements.

Table 2: Inter-process primitives for each connection type when the source and destination components are allocated to the same processor

Connection type	Data	IPC primitives
Synchronization	Yes No	Blocking mailbox Binary semaphore
Asynchronous data transfer	Yes (link is ignored if there is no data)	Non-blocking mailbox

To achieve this, the first step is to ensure that the run-time tasks execute sequentially or concurrently, as specified by the model. As shown in Fig. 7, each run-time task is divided into three phases to support the concurrency control mechanism as follows.

- Growing phase: when a task is about to be invoked, it reserves locks for all synchronous inputs. If asynchronously transferred data is already available, it is accepted from a buffer using the non-blocking IPC primitives shown in Table 2. If it cannot immediately acquire all the required inputs, it waits until it can (using blocking IPC primitives).

- Task execution phase: software components are executed in the context of the run-time task, as defined in Section 3.2.1.

- Shrinking phase: when the execution of a task context is complete, it releases all of its locks and sends all data to its output connections.

Note that the priority ceiling protocol (PCP) is used, if necessary, to combine synchronization with the concurrency control mechanism to prevent deadline overruns.

By traversing the graph that represents an application run-time model, synchronization and asynchronous data-transfer requirements are identified for both the input and the output of each run-time task. Because global variables are required to initialize and use the IPC primitives, we maintain a table of global variables, which are numerically indexed, with names such as `Sem_0` and `DataMbox_0`.

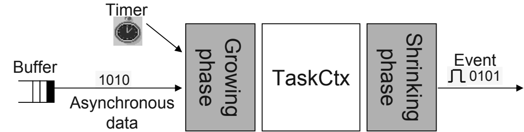


Figure 7: An example concurrency-control mechanism for run-time tasks

If the source and destination of a link are not assigned locally to the same processor, time- and event-triggers will be required, and additional steps may be needed to generate the necessary code. When a time-trigger is activated by a specified time clock, the RTOS must determine which run-time task is to be triggered. Each instance of a time-trigger in the timer interrupt handler (ISR) generates a global variable such as `Timer_Task0`, and uses the IPC primitives for synchronization during the growing phase of a run-time task which does not involve any data transfer.

An external message cannot use a global variable in this way because it is not valid outside the sending processor. Each external message is therefore uniquely indexed throughout the entire system. This approach has already been used in some existing communication devices. For example, the controller area network (CAN) device identifies the messages that it receives, using the message ID field [17]. This type of communication can be easily implemented by adding a message index to the network header or to the message content, and then routing each message by passing its index to the interrupt handler of the communication device.

In the shrinking phase, sending data and messages through an external communication device is no longer handled by the IPC primitives. Instead, an API for the device driver of the communication device is used to send messages to the destination processor as soon as the external message index and the data content become ready.

If virtual memory management is used in a processor with a memory management unit (MMU) [15], the address space for each process is usually isolated from others and is invisible to other processes. This implies that direct references to local variables stored in the stack of another run-time task are not possible. Run-time links between two

tasks must therefore be handled very differently from links within a task. In Section 3.2.1, we allocated local variables for the links that transfer data within a task. A similar requirement applies to links between different tasks, but local variables are now useless because they cannot be accessed by both the source and the destination components, or shared between a component and the device driver API that is used to send data. We therefore allocate variables in a globally-accessible memory area, or heap, via real-time dynamic storage allocation (DSA) [5].

3.2.3. Time- and Event-Triggers for Run-Time Tasks

Our system model contains a mixture of event- and time-triggered components to reflect the increasing use of external devices with a high degree of temporal uncertainty in distributed real-time systems [14].

As described in Section 2.3, each run-time task is triggered by a timer, if a fixed invocation rate is assigned, by an event which is generated by an output component in another transaction, or by the arrival of data at an external communication device. Nevertheless, we remain to assume that our target RTOS is based on conventional preemptive priority-based scheduling, like most existing systems.

We implement each time-trigger with a release offset and an invocation rate, using a fine-granularity periodic timer. This can be a dedicated hardware timer or a hook for the OS tick timer. During each time period, an interrupt handler for the timer is activated and looks for expired time-triggers. If any, the handler uses a pre-defined timer table to notify the corresponding run-time tasks.

Each event-to-task link in the application run-time model is abstracted to be an external message which is defined by its message index (as described in Section 3.2.2) and its timing characteristics, namely, release offset and invocation rate, both of which are inherited from the output of the component that generates the message. In our system, a communication device is modeled as a peripheral which generates an external message with its message index at a specified time clock, using the timing characteristics provided by the application run-time model. To avoid any loss of data, each software component must pre-allocate enough memory for a buffer to which the interrupt handler in the RTOS can copy data received from the communication device before dealing with it.

3.3. System Monitor

Our framework, which is integrated with the existing AIRES toolkit, provides a more accurate and more efficient way of supporting high-level decisions for design and verification of distributed real-time systems. As shown in Fig. 5, an application run-time model is composed of an application structure model and computation and communication resource information, and is automatically converted to a form that is executable on the virtual execution platform, by automatic generation of glue code. We have also developed a timing and resource monitor to collect and analyze run-time performance data. This provides functions, such as the ability to trace the time at the entry to and exit from software components, to generate and receive messages between run-time tasks, and to monitor, as well as other events as shown in Fig. 9. This enables us to refine the whole system model, comprising hardware, RTOS, and the application, efficiently at successive design levels.

On the other hand, by appending an exception handler to the end of the growing phase or to the beginning of the shrinking phase in each run-time task, we can capture the events of deadline misses in time-driven transactions. When a deadline miss occurs, a user-defined error-handling function is invoked to take an appropriate action, such as shutting down the system for safety.

The memory allocated by real-time DSA for data transfer among run-time tasks is freed explicitly in the destination component or in the device driver after the data has been used. This means that the memory requirement varies with the number of external links, and the system monitor must check for memory overflow, using an exception handler integrated into the memory allocator.

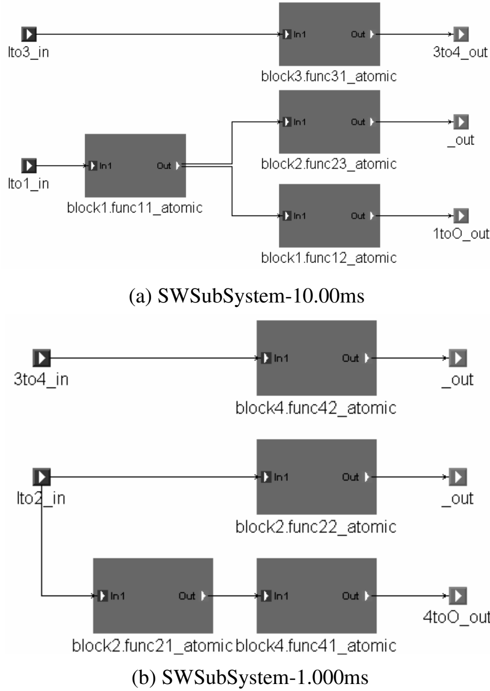
4. Evaluation

Our integrated framework is applied to a simple automotive control system illustrated in Fig. 8. The hardware system composed of two processors, Proc1 and Proc2, which are connected by a single CAN bus. The timer resolution is set to 0.1ms. The run-time properties and the timing constraints of the application run-time model are summarized in Table 3. In the application run-time model, all the links are synchronous except for the link between `func31` and `func42`, which is a non-blocking data transfer using `DataMbox_1` as a global variable.

Table 3: Run-time properties of a simple control system

Run-time task	Release offset	WCET	Deadline	Period	Priority	Allocated processor	Software components
Task1	0.594	0.024	1.000	1.0	4	Proc1	`func41`
Task2	0.478	0.026	1.000	1.0	6	Proc1	`func42`
Task3	0.000	0.024	0.478	10.0	8	Proc1	`func31`
Task4	6.034	0.025	10.000	10.0	2	Proc1	`func12`
Task5	0.000	0.038	6.034	10.0	3	Proc1	`func11`
Task6	0.000	0.025	1.000	1.0	5	Proc2	`func22`
Task7	0.000	0.028	0.461	1.0	7	Proc2	`func21`
Task8	6.034	0.024	10.0	10.0	1	Proc2	`func23`

We collected the run-time behavior and performance data of Proc1 using the system monitor described in Section 3.3, which runs a simple control system application to capture the timing characteristics. As shown in Fig. 9, the sys-



(a) SWSubSystem-10.00ms

(b) SWSubSystem-1.000ms

Figure 8: Software structure of a simple control system

tem monitor displays the states and interactions at the hardware, RTOS, and application levels. The timing information for each run-time task is further decomposed into the growing phase, the task context, and the shrinking phase, so that we can trace each synchronization and asynchronous data-transfer primitives which are described in Section 3.2.2.

To verify the system-level end-to-end timing constraints on the control system application, we traced the time at which the shrinking phase is finished for each run-time task to obtain the worst-case response time (WCRT). The values of WCRT measured on the virtual execution platform and listed in Table 4, together with the deadline defined in the application's non-functional requirements in Table 3, demonstrate that the run-time tasks that our system has constructed, do not violate their deadlines. Nevertheless, we observe that the values of WCRT obtained by real-time analysis do not correspond to those measured on the virtual execution platform. This may be due to additional overheads incurred by the RTOS kernel and interrupt service routines, which are not fully considered in the real-time analysis techniques.

It is well-known that reducing the granularity of the timer increases the overhead for timer interrupt handler while the interval jitter of each timer is highly dependent on the timer clock resolution. So, we performed more simulation while varying the timer resolution, to see how it affects the response time of a run-time task. For this purpose, we selected the WCRT of Task1 on Proc1 as an example, and measured its value while changing the timer resolution from 0.5ms to 0.011ms. As shown in Fig. 10, the CPU utilization increases linearly from 17.6% to 96.3%. Note that changes to a parameter, the timer resolution in the experiments, can make significant impact on the overall system performance.

Regarding the system-level end-to-end timing constraints, the timer resolution must be in the range of 0.333ms to 0.013ms, so that the response time of Task1 does not violate the system-level deadline. To accommodate software components for the run-time tasks allocated to Proc1, 5.8% of the CPU time on Proc1 (calculated by a real-time analysis) is reserved. However, as shown in Fig. 10, at least 12.8% more CPU time will be consumed by the RTOS to meet all the non-functional timing constraints on a virtual execution platform.

Table 4: Comparison of worst-case response time (Proc1)

Run-time task	RT-analysis WCRT	Simulated execution platform WCRT	Simulated execution platform Average RT
Task5	0.118	0.245	0.232
Task4	6.133	6.215	6.215
Task3	0.023	0.146	0.134
Task2	0.503	0.627	0.625
Task1	0.667	0.724	0.723

To apply the integrated framework effectively for the development of large-scale distributed real-time systems, a virtual execution platform must be generated fast enough to support frequent modifications, such as changes to tuning parameters, while dozens of distributed processors might be used to verify the system-level non-functional requirements. Also, the time needed to simulate the system must not be too large. We, therefore, measured the time needed to generate a virtual execution platform for the simple control application mentioned above, as well as for an automotive application, GenericVSC [11], which has a larger number of software components and links. The experimental results are summarized in Table 5. The generation time for one software component is less than 22ms for the simple control application. It appears that most of the time is taken by initialization, compared to between 3 and 5ms for GenericVSC. The virtual hardware simulator runs for 100s on top of a window-based PC to simulate 9.410s in real time while it executes 423974308 instructions of the executable binary for the control application. This suggests that the integrated virtual execution platform is fast enough to support the design and verification of large-scale systems.

5. Related Work

Several recent projects dealt with RTOS modeling for embedded system design. Most of this research [7, 8, 13] involved the use of transactional modeling tools intended

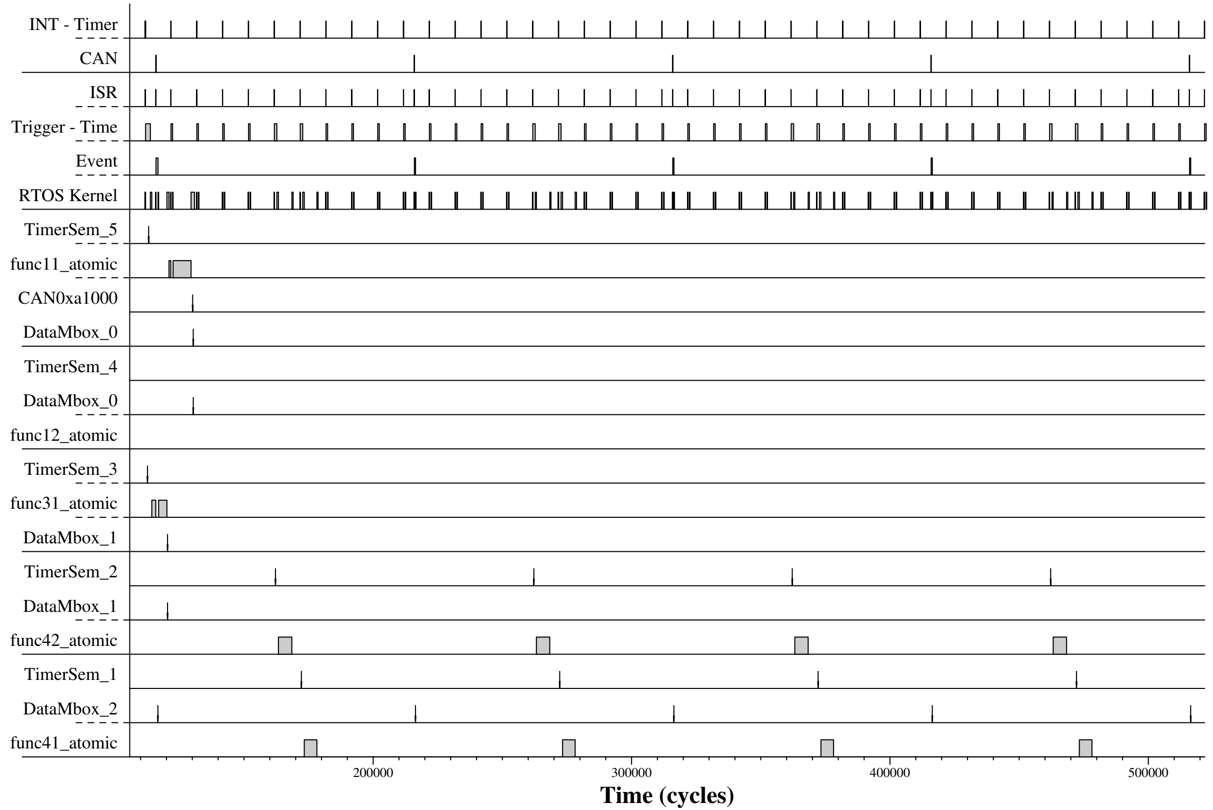


Figure 9: An example of run-time behavior and performance data for a simple control system on the virtual execution platform (Proc1)

Table 5: Time required to generate the virtual execution platform for two applications

Application		Simple control	GenericVSC
Num. of software components		8	56
Num. of links		15	1280
Generation time (ms)	Proc1	171	188
	Proc2	172	234

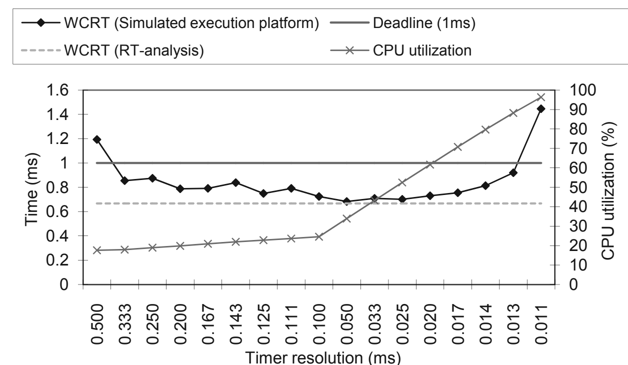


Figure 10: The dependency of WCRT and CPU utilization on the timer resolution (Task1)

for design of hardware systems, such as SystemC [16], to model the behavior of the RTOS and the time delays that it incurs, as well as to simulate task scheduling. Either a dedicated RTOS is generated, or glue code is used, to assemble a system which has one common description language for both hardware and software. Compared to traditional hardware design techniques, this approach simplifies the design and verification of systems that run application software in the form of concurrent tasks that require synchronization.

However, hardware description languages have a limited ability to model the dynamic behavior of real-time systems, and moreover, some procedures must be coded manually. Hessel *et al.* [9] extended the SystemC syntax to support dynamic behavior, and applied their abstract RTOS model to refine the real-time scheduling of tasks.

Research on hardware description languages has largely focused on accurate and fast co-simulation of software and hardware by modeling the RTOS, which is the underlying support software, rather than on validating non-functional constraints, such as the end-to-end timing delay and resource capacity. To develop modeling tools useful in real-time systems, non-functional constraints must be included

in the model, using techniques such as scheduling refinement, while satisfying timing constraints [9] and performing schedule monitoring and stack safety analysis [2].

Henriksson *et al.* developed the TrueTime [4] toolkit to monitor the temporal uncertainty that occurs when distributed control loops are implemented as tasks in a real-time kernel and communicate with other processors over a network. Our approach is similar to TrueTime in that it focuses on support of high-level design decisions with emphasis on non-functional constraints. However, we construct a more detailed and more accurate model that describes a system from bottom to top, including the application, RTOS and hardware, whereas TrueTime uses an abstract RTOS and network model.

6. Conclusion

In the development of large-scale distributed real-time systems, non-functional constraint problems have been addressed piece-by-piece: for example, by code-level optimization of the software on a running system. In the component-based software development, the need to satisfy multiple constraints becomes a high-level decision problem within the system composition process. To make good high-level decisions of this sort economically requires a development toolkit that is capable of obtaining and analyzing accurate run-time behavior and performance data.

To achieve this, we have developed a framework for generating a virtual execution platform on which the system can be run so as to obtain accurate run-time performance data. This framework has been implemented and integrated with the existing AIRES toolkit. Our preliminary evaluation has shown that changes to a few parameters in the system design may make significant impact on the overall performance. By running the entire system we were able to collect representative information for quantitative analysis of the target system. This integration of the application run-time model with the execution platform will help the system designer understand the performance of a system in a more holistic fashion, and permit systems to be developed in a more convenient and efficient way.

Acknowledgements

This work was supported in part by Hitachi America, the Escher institute, and VaST Systems.

References

[1] AIRES. http://kabru.eecs.umich.edu/bin/view/main/aires.

[2] M. Briday, J.-L. Bechennec, and Y. Trinquet. Task scheduling observation and stack safety analysis in real time distributed systems using a simulation tool. In *Proc. of the IEEE Conference on Emerging Technologies and Factory Automation*, pages 299–306, 2005.

[3] K. bun Yue, S. Davari, and T. Leibfried. Priority ceiling protocol in Ada. In *TRI-Ada '96: Proceedings of the conference on TRI-Ada '96*, pages 3–9. ACM Press, 1996.

[4] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. E. Arzen. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, 2003.

[5] A. Crespo, I. Ripoll, and M. Masmano. Dynamic memory management for embedded real-time systems. In *Proc. of the IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 195–204, 2006.

[6] J. A. Darringer, R. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J. Morell, I. I. Nair, P. Sagmeister, and Y. Shin. Early analysis tools for system-on-a-chip design. *IBM Journal of Research and Development*, 46(6):20–38, 2002.

[7] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS modeling for system level design. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 130–135, 2003.

[8] Z. He, A. Mok, and C. Peng. Timed RTOS modeling for embedded system design. In *Proc. of the IEEE Real Time and Embedded Technology and Applications Symposium*, pages 448–457, 2005.

[9] F. Hessel, V. M. Rosa, C. E. Reif, C. Marcon, and T. G. Santos. Scheduling refinement in abstract RTOS models. *ACM Trans. on Embedded Computing Systems*, 5(2):342–354, 2006.

[10] C. Krishna and K. G. Shin. *Real-Time Systems*. McGraw Hill, 1997.

[11] J. R. Merrick, S. Wang, and K. G. Shin. Priority refinement for dependent tasks in large embedded real-time software. In *Proc. of the IEEE Real-Time and Embedded Technology and Application Symposium*, pages 365–374, 2005.

[12] Micrium. uC/OS (http://www.ucos-ii.com/).

[13] R. L. Moigne, O. Pasquier, and J. Calvez. A generic RTOS model for real-time systems simulation with SystemC. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 30082–30087, 2004.

[14] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 119–126, 2004.

[15] A. SilberSchatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts*. Wiley, 2000.

[16] S. Swan. An introduction to system-level modeling in SystemC 2.0. Technical report, Cadence Design Systems, Inc., 2001.

[17] Texas Instruments. Introduction to the controller area network (CAN), 2002.

[18] VaST Systems. VaST tools and models for embedded system design (http://www.vastsystems.com/).

[19] S. Wang and K. G. Shin. Task construction for model-based design of embedded control software. *IEEE Transactions on Software Engineering*, 32(4):254–264, 2006.