EXPLORING THE DESIGN SPACE OF IMA SYSTEM ARCHITECTURES

Richard Bradford, Shana Fliginger, Rockwell Collins, Cedar Rapids, IA
Min-Young Nam, Sibin Mohan, Rodolfo Pellizzoni, Cheolgi Kim, Marco Caccamo, Lui Sha
University of Illinois at Urbana Champaign, Urbana, IL

Abstract

In systems such as integrated modular avionics (IMA), there is a substantial benefit from maintaining significant portions of a product family's architecture unchanged from one system to the next. When there are tight constraints on resources such as bandwidth and processor capacity, however, certain seemingly small changes in a few components have the potential to create a cascade of timing problems. The ability to rapidly analyze and quantify the impact of these changes prior to implementation and system integration provides the engineering team with early validation of the changes, which can prevent substantially increased costs for design, integration, and verification, as well as delays in the development schedule.

However, detailed early evaluation of architecture performance involves analysis of many complex interrelated variables and is therefore challenging. Consider the case of moving a task from a processor's IMA partition to another processor's partition. The tasks sets need to be updated. The I/O and network traffic must be rerouted. The schedulability equations of processor, I/O and network need to be recreated, and the analysis needs to be propagated end to end. Last but not least, all the architecture specification documents have to be updated.

In order to reduce the detailed architecture evaluation effort, we have automated the performance analysis process with a system integration tool prototype called ASIIST (Application-Specific I/O Integration Support Tool). To move a task, we can now use a graphical interface to drag and drop a task from one processor's IMA partition to another. All the steps described above are done automatically, including the updating of the architecture specification in AADL (Architecture Analysis and Description Language). In this paper, we show how to use this tool to explore the design space of an IMA system architecture, so as to derive designs with specified performance properties.

Introduction

The introduction of integrated modular avionics (IMA) has brought increased computing power and memory into the designs of avionics systems. A typical IMA architecture may include several computing platforms, each hosting multiple software applications, in addition to various specialized processing components, all connected by fault-tolerant real time networks. These architectures give rise to many inter-related decisions that must be made by system architects, such as how to assign software components to hardware processors, how to schedule the processors to accommodate the applications' execution time requirements, and how to allocate network bandwidth to message traffic to achieve latency requirements. Because of the complexity of this decision-making process, as well as the potential savings available with component reuse, there is a substantial benefit from maintaining significant portions of a product family's architecture unchanged from one system to the next. When there are tight constraints on resources such as bus bandwidth, network bandwidth and processor capacity, however, certain seemingly small changes in a few components have the potential to create a cascade of timing problems. The ability to rapidly analyze and quantify the impact of these changes prior to implementation and system integration provides the engineering team with early validation of the changes, which can prevent substantially increased costs for design, integration, and verification, as well as delays in the development schedule. The management of the various architectural decisions involved in complex system design and development often requires significant effort and schedule time to perform with sufficient detail. System designers would benefit from automated tool support to maintain the desired degree of control of complex system designs as individual components' designs and requirements evolve. In particular, a system integration tool should provide indications of performance problems as early as possible. However, detailed early evaluation of

978-1-4244-6618-4/10/$26.00 ©2010 IEEE

architecture performance involves analysis of many complex interrelated variables and is therefore challenging. Consider the case of moving a task from a processor's IMA partition to another processor's partition. The tasks sets need to be updated. The I/O and network traffic must be rerouted. The schedulability equations of processor, I/O and network need to be recreated, and the analysis needs to be propagated end to end. Last but not least, all the architecture specification documents have to be updated. Real time performance analyses of processor loading and I/O latency have traditionally been, to a large extent, done independently, requiring significant effort to consolidate the information into overall results to support design decisions during early stages when multiple teams are continually making changes to the design.

This paper presents an automated system for analyzing architectures to achieve designs that allocate system resources efficiently while meeting requirements for latency and for temporal coordination. It combines a simple design pattern that enables a synchronous design to be distributed over asynchronous components with modeling and analysis techniques that ensure the design satisfies the system resource constraints. We have automated the performance analysis process with a system integration tool prototype called ASIIST (Application-Specific I/O Integration Support Tool). To move a task, we can now use a graphical interface to drag and drop a task from one processor's IMA partition to another. All the steps described above are done automatically, including the updating of the architecture specification in AADL (Architecture Analysis and Description Language). We show how to use this tool to explore the design space of an IMA system architecture, so as to derive designs with specified performance properties.

In the next section, we describe the various decision problems that the modeling tool helps the system designer solve. Following that, we describe the analytical approaches that are embodied in the modeling tool, and we examine a simple numerical example. Finally, we discuss related work and summarize our conclusions.

Decision Problems in System Design

As described above, the process of architecting, designing, developing and integrating an IMA system presents unique challenges to system developers, who must allocate system resources to numerous subsystems and components and must ensure that components executing asynchronously establish a sufficient degree of coordination to maintain logically correct behavior. In this section we describe the various types of analysis and support that ASIIST provides to inform the decision-making process of the IMA system architect.

Application Scheduling

When multiple applications share a processor, the system architect must allocate the processor capacity to each partition in an IMA so that each application receives sufficient processing time to complete its tasks by their deadlines. ASIIST uses a scheduling algorithm [1] that uses task periods and processing requirements to determine each application's required processor allocation at both the IMA partition level and at the task level within each partition. System architects can often determine appropriate task periods based on functional requirements, but determination an application's processing requirements will often require a more detailed analysis of worst-case execution time (WCET).

Worst-Case Execution Time (WCET) Analysis

The ASIIST tool chain includes a timing analyzer that derives safe upper bounds on the WCET of given code segment. As discussed later, this timing analyzer contains models of several commonly used processors.

Sometimes a system architect must make assessments very early in the development process about the feasibility of a particular system architecture. In many cases the results of these assessments depend crucially on the WCET of a new or enhanced software application, and unfortunately, the system architect's feasibility assessment is sometimes required before the software is actually written. The ASIIST tool chain supports the early assessment of architecture feasibility [2] based on timing analysis of code generated automatically from functional models developed in Simulink®.

When multiple applications run together on a cached CPU, interference from communication flows on the front side bus (FSB) must be considered: since main memory is a shared resource, when a CPU cache controller tries to fetch a cache line after a cache miss, it can be delayed by a peripheral reading/writing in memory. This delay can significantly increase the application's actual WCET. ASIIST uses the algorithm introduced in previous work [3] to compute the maximum cache delay and thereby derive a new modified WCET for the task.

Bus Delay Analysis and Cache Interference

A logical data flow may traverse multiple hardware components and/or a network in between its source and destination applications. Depending on the type of hardware used, data traffic may be regulated according to different arbitration schemes and COTS protocols. A designer can also use various I/O configurations to improve system performance. System performance is dependent on hardware flows that are derived from the logical data flows and I/O configurations. To specify such hardware flows, ASIIST allows the user to define properties in AADL to represent the various possibilities that could occur for the same logical data flow.

Based on the hardware flow specification, ASIIST can automatically apply delay analysis, based on the theory of network calculus [4], to compute safe delay and buffer bounds for a variety of possible components and arbitration schemes.

Using this methodology, system architects can derive quantitative assessments of alternate hardware designs and quickly verify their feasibility at an early stage. As an example, the authors of [5] proposed the use of real-time bridges, unintrusive devices interposed between COTS peripherals and buses, to predictably control data flows. ASIIST supports the integration of this I/O management system and can compute improved delay bounds when real-time bridges are employed.

Logical Synchrony

Individual nodes of a distributed system inherently operate asynchronously. To achieve correct logical behavior, then, designers of real-time systems must implement protocols to enable the individual nodes of these systems to agree on the shared subset of the global system state. Development and verification of applications that achieve functionally correct, fault-tolerant, real-time results can be extremely challenging. To facilitate this process, ASIIST now supports the use of the Physically Asynchronous/Logically Synchronous (PALS) design pattern [6]; it has been shown that this pattern allows developers to design and verify a system as though all nodes executed synchronously, which is an immensely simplifying abstraction [7].

Support for Architectural Patterns

ASIIST provides support for the use of software architecture design patterns in a straightforward way. First, ASIIST will automatically insert scheduling overhead values from design patterns in AADL specified and annotated (by AADL properties) by the users. Second, ASIIST users may impose design rules and check whether annotated tasks follow the rules. We will illustrate this using the architecture design pattern known as Physically Asynchronous Logically Synchronous (PALS) [6,7]. The PALS design pattern implements real time logical synchrony over a networked control system with bounded asynchrony caused by the skews between distributed clocks. PALS allows developers to design and verify a distributed system as though all nodes executed synchronously, then distribute this design over a physically asynchronous architecture following a few simple constraints that ensure that the logical correctness of the synchronous design is preserved. PALS also provides the fastest possible rate to achieve consistency in distributed views and actions. In a study of implementing PALS in an Integrated Modular Avionics system, PALS reduced the time to formally verify the correctness of an active standby design from 35 hours to less than 30 seconds [7].

A library code for PALS has been developed for easier application of the pattern to existing software code; the library forms a programming interface. Message packets transmitted at a local clock tick are delivered at the next clock tick if PALS is used. One of the important benefits of PALS is the consistency from this determinism in communications. The library exploits this consistency, providing consistent views on the global system states among the tasks. Hence, the library substitutes the read/write interfaces of global system access for communication

interfaces. The overhead of the library can be statically analyzed.

PALS is used for tasks that need global consistency, called global computation, e.g., replicated nodes for fault tolerance. All the communication between them must be defined through the library, and overhead will automatically be added by ASIIST. Once tasks involved in global computation are annotated, we can easily check if they follow this design rule. In addition, if users intend to communicate with global computation tasks, this communication must go through the library's input and output services respectively. This design rule can also be easily checked and enforced.

Modeling and Evaluation of Patterns

Using software architecture patterns, such as PALS, may add cost through increased execution time of the processor, which must be accounted for by the schedulability analysis. In order to allow the use of such architecture patterns as an early design choice, the extra execution time needs to be quantified. Depending on the pattern, any parameter that may define the variance or the worst case bound of the added execution time must be specified to determine the before and after schedulability of the set of applications which may use the pattern or not. We call these additional times *wait states*. Wait states are newly introduced extra times that are added due to optional features or external I/O of the system that was not or could not be considered during source code timing analysis.

For the case of PALS, PALS period, maximum clock jitter, and maximum communication delay of all logical data connections needs to be specified for each AADL thread that would use the PALS library. Max publishing time per variable, max reception time and basic PALS's library overhead time are described for each AADL process. Some PALS parameters, such as the maximum communication delay and basic library overhead, are used in constraints that can be checked by ASIIST. ASIIST acts as a requirements checker when time-related assumptions exist for patterns. A thread that has been previously configured to use PALS with a period of 10 ms may not be realizable when the application is moved to a new processor with greater I/O traffic due to its increased communication delay. Any architectural requirements such as "all PALS threads should only communicate with other PALS threads through data ports" are checked by ASIIST for correct usage of patterns.

Virtual Integration and Analysis for IMA

In this section we present the concept of *virtual integration*. The main idea is that all the necessary analysis and estimation of system integration issues can be performed *ahead of time* using models of the system -- both hardware as well as software models. For IMA systems where multiple groups /subcontractors must work together to build complex safety-critical systems, such early and rapid analysis can significantly reduce the amount of time and effort spent during later (system integration) stages. The virtual integration *framework* that we have created (Figure 1) consists of the following steps/tools: *(a)* modeling of functional behavior; *(b)* timing analysis to determine worst-case execution times (WCETs) and *(c)* schedulability and analysis of bus delays.

As mentioned above, we create software as well as hardware models for our analysis. The software models are created using Simulink® while the hardware models are created using AADL. Simulink is a useful tool in that we are able to model functional requirements at varying levels -- either in precise detail or at abstract levels. It also has a fairly mature code generation capability. This is necessary since most contemporary timing analysis frameworks require access to the code at some level -- either the source code or assembly/binary code. Hence, the use of Simulink overlaps the first two stages of the virtual integration framework described above -- the modeling of functional behavior and the necessary code generation for the timing analysis stage.

If the analysis framework depicted in Figure 1 confirms that an application is schedulable, then it is very likely that a final (optimized) implementation is also schedulable. This is due to the fact that every step in our analysis is conservative in nature -- part of this conservatism is intrinsic in the modeling of the hardware itself (since the behavior of many COTS components is regulated by a complex set of parameters that are not easily specifiable at design time). The final implementation will have software/hardware that is optimized thus ensuring that it performs better than the conservative models we use. Thus, assuming that the models were correct, the analysis is guaranteed to produce *safe bounds*.

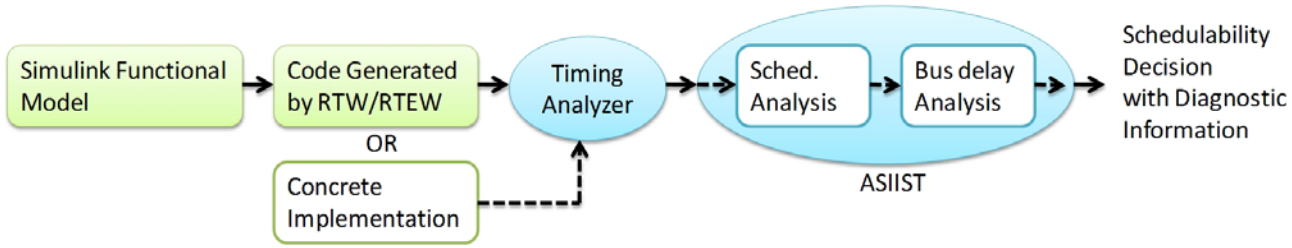

Figure 1. Overview of the End-to-End Analysis Framework

The following sections provide details about the timing analysis phase and discuss schedulability and delay analysis using our ASIIST tool. Later sections provide details about the modeling of hardware components (processing elements, memory and bus architectures, etc.) using AADL.

Timing Analysis and Code Generation

Timing analysis, to determine the WCET of the application, is performed by use of our static timing analysis framework [8-10]. A compiler front-end takes C source files as input and extracts control flow and constraint information. A static cache simulator extracts information on whether instructions will hit or miss the cache. All of the above information along with machine dependent information (pipeline stages, instruction set, etc. is fed to a *timing analyzer* (TA) that derives WCET bounds. These WCET values are *safe* since they are conservative in nature and include overestimations of the execution time. *Note*: the TA calculates the WCET without information about program inputs. The timing analysis framework is shown in Figure 2.

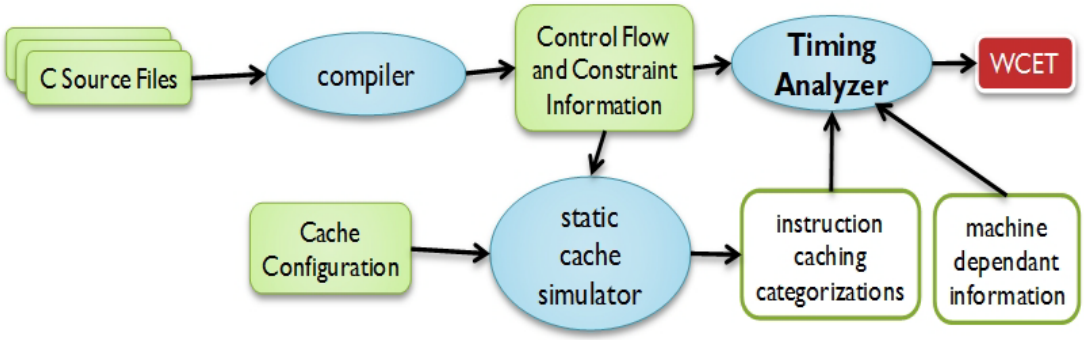

Figure 2. The Static Timing Analysis Framework

The TA framework has the ability to handle different processor models, *viz*., MIPS, SPARC and Atmel. It can also analyze different instruction cache configurations. In our experiments we used the MIPS processor model and the PISA instruction set and an instruction cache that is 4-way set associative and 64k in size. We did not consider data caches and assumed that all data references go to memory, but existing or new data cache analysis techniques can be easily integrated into this framework. As shown in Figure 1, the TA can either take in the source files generated by Simulink or an actual concrete implementation of the code if it exists.

Note: While we use static timing analysis techniques for obtaining the WCET values for the code generated from Simulink we realize that not all code used in the system may be amenable to static analysis. This could be due to the complexity of the code, its dependence on various libraries or even the necessity for user/environment-based inputs. Hence, system architects may sometimes need to *profile* the execution behavior of the code and derive WCET values using available worst-case input values. The virtual integration framework (and relevant tools) presented in this paper can still be used; in fact, in this work, we used profiling techniques based on worst-case input values to obtain the WCET values for the PALS middleware library.

Timing Analysis for PALS Middleware Library

As mentioned in the previous subsection, we used profiling-based techniques to determine the worst-case behavior of the PALS middleware library. We used the following procedure to obtain the execution times for the middleware library: *(a)* used the *Klee Symbolic Virtual Machine* [11] to exhaustively analyze the source code and obtain an exhaustive list of all execution paths and input values that drive these execution paths; *(b)* executed the library and core logic components using the generated input sets to measure the execution time for each path and *(c)* analyzed the measured execution times to obtain the WCET for the library/logic code.

We executed the code (with generated inputs) on a PC104 embedded platform that has a 1.6 GHz Intel Atom processor. The library was analyzed along three categories, split along functional lines: (1) Library Initialization Phase, (2) Core Logic execution + Library Overhead, and (3) Packet Reception Overhead

The first phase sets up the library for the first time with various parameters, etc. – this phase is executed just once. The second phase is the execution of the core logic, *i.e.*, the part that implements the PALS protocol. This part is executed every time distributed synchronization is required, *i.e.*, every cycle. The final phase calculated the overhead that is incurred when a packet is received. This final value occurs every time some communication takes place.

The Library initialization phase has just **one** path, and we executed it with **10** different inputs values suggested by Klee. We obtained an upper bound of **233 µs** for this phase -- the third input set resulted in this execution time value, and this was the largest of the 10 experiments.

The Core logic and Library execution phase has a total of **349** distinct paths. Each of these paths was also executed with **10** distinct input sets generated by Klee. The execution times for all **3490** experiments are shown in Figure 3. The x-axis lists the various experiments while the y-axis depicts the execution time in microseconds. As we see from this figure, the largest execution time of any path is obtained from experiment number **3111** (*i.e.* the first run of path **311**). Hence the WCET value is **126.90 µs**.

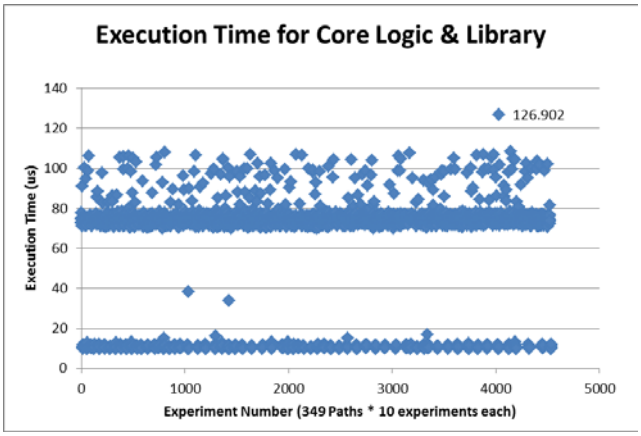

Figure 3. Execution Time for PALS Core Logic and the Library

The final phase *i.e.* estimation of the overheads for packet reception has **14** distinct paths – we executed each of these paths **10** times each based on inputs generated by Klee. The largest execution time obtained for this phase was **81.714 µs**.

The overhead for measurements was **1.816 µs**. As we see from this value, this is a very minor addition to the execution times of each of the experiments.

Schedulability Analysis

The WCET information obtained from the TA framework is used for the schedulability analysis performed by ASIIST. The WCET of each application is measured in isolation, assuming no contention for access to shared system resources such as communication infrastructure and memory elements. In practice, modern embedded architectures employ multiple active components that can autonomously start data transfers. Peripherals that need to exchange high throughput traffic such as video flows usually have DMA capabilities. In a multiprocessor system, other tasks can run simultaneously with the application under analysis.

In particular, contention for access to main memory can greatly increase the WCET of an application running on a cached CPU. When a CPU cache controller tries to fetch a cache line (after a cache miss), if other CPUs or peripherals are trying to access main memory at the same time, the fetch operation can be significantly delayed. In turn, the application will experience delays thus increasing its WCET from the estimated values. In our experiments

[12], the WCET values were increased as much as 196% for a system including two processing cores and a single peripheral.

To address such issues, ASIIST is able to compute worst case upper bounds on memory contention wait states and use them together with the provided WCET in isolation to perform extensive schedulability analysis. Our memory contention analysis is based on information about the tasks running in the system and peripheral traffic to/from main memory. Tasks are divided into sequential scheduling intervals; WCET in isolation and maximum number of cache misses must be provided for each interval.

We also assume that a bound on the total amount of traffic to/from main memory is known (calculated by the bus delay analysis functionality of ASIIST). Given all of this information, our algorithms [12, 13] compute the maximum wait states in each scheduling interval and hence derive a new modified WCET for each task. ASIIST then applies the schedulability algorithm [1] to calculate the schedulability of all the applications that share the CPU in an IMA system. More details about ASIIST and its capabilities are provided below.

Bus Delay Analysis and Solutions

Standard COTS interconnections such as PCI and PCI-Express are becoming commonplace in embedded computational nodes. A logical data flow can go through multiple hardware components to get to a destination. For example, the PCI interconnection [14] has a tree structure where peripherals transmit on bus segments connected by bridge components. In PCI-Express, processing elements, memory elements and peripherals are connected by point-to-point links and packet switches. Multiple bridges/switches can improve the communication parallelism in the system, but they increase communication delay and can severely impact end-to-end application deadlines. ASIIST allows the designer to quickly specify different interconnection configurations and explore relevant trade-offs in communication bandwidth, delay and impact on memory contention.

In particular, we previously showed [15] how ASIIST can be used to derive delays for data flows on the PCI bus. Our employed analysis methodology is based on the theory of network calculus [4] that is flexible enough to model a wide variety of arbitration models and buffering schemes. Initial delay bounds can be obtained using general assumptions, although they can be pessimistic. The pessimism in the analysis depends both on the complex behavior of low-level COTS arbiters, and on the variances of DMA peripherals timing behaviors. By refining the model during the design process the user can obtain tighter bounds, at the cost of specifying additional architectural details. If the designer is willing to pay some additional hardware cost, then the I/O management system that will be introduced in a later section can significantly improve the design process by enforcing transmission budgets for peripherals and isolating I/O flows with respect to critical software partitions.

Modeling and Tool Support by ASIIST

The requirement for early analysis motivates the need for some sort of input for analysis that represents a system. Architecture Description Languages (ADLs) are popular for this purpose, and we use SAE AADL (Architecture Analysis and Design Language) [16] which is increasing in use due to its well-defined semantics for real-time embedded systems and standardization. Other languages such as SysML and MARTE are also popular and thus are sometimes used together to compensate each other.

One of the benefits of using AADL is that it allows software and hardware components to be modeled separately with strong semantics. Applications are bound (by AADL properties) to hardware components to represent whether certain threads are executed on a processor. A list of hardware components can be used to represent the hardware data path for logical data connections among applications. Such standard properties which are commonly used are defined for the AADL standard. New properties can be defined by the user for other purposes in cases where new abstractions that are not defined in the standard are needed.

Application Specific I/O Integration Support Tool (ASIIST) [15] is a software tool which takes AADL models as input and mainly performs schedulability analysis and bus delay analysis for the model. It focuses on incorporating the side effects between these analyses. Memory contention and I/O wait times are examples of how I/O flow will affect

the schedulability of tasks. Using the PALS library would affect the schedulability and the delay of data because of the extra middleware code and the pre-defined PALS period. Also, changes in the hardware processor or bus topology will change schedulability and end-to-end latency, which is very difficult to re-evaluate without a tool like ASIIST.

In order to correctly capture the effect of I/O traffic, we require the user to add I/O configuration data to the model. Each logical connection would have multiple hardware lists that represent the actual I/O configurations used for the logical connection. Figure 4 shows an example where data is received through the peripheral to be read from the processor. We assume that the application that requires this logical connection is currently assigned to partition A. In this case, the I/O configuration is set so that the data is initially stored in the local memory and later read from the local memory. The user would specify the write bus transaction (f3) and specify the read bus transaction (f1, f2) to represent such a configuration. Since the read transaction is initiated by the application, the delay will create a wait state for the application. In an IMA system, f3 will exist across each partition because there is no constraint on the peripheral. ASIIST would duplicate such flows automatically for each partition and compute the worst case delay for each logical connection. It will also add the wait states produced by I/O wait times to the execution time for schedulability analysis.

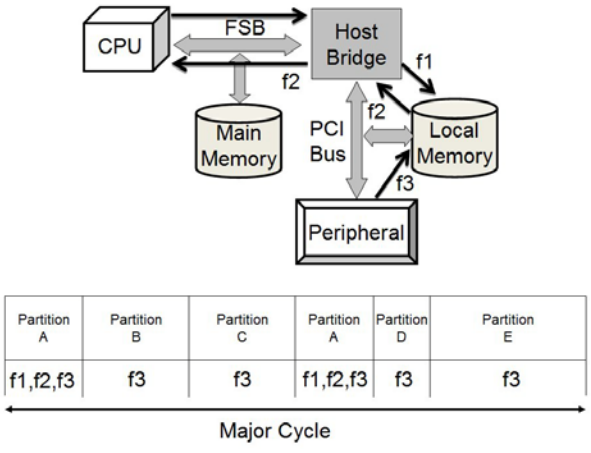

Figure 4. I/O Configuration and Side Effect

Other choices of I/O configurations would be to have the data be directly read from the peripheral or to have the peripheral directly send the data to the main memory with DMA. In the former case, the I/O wait time would increase but would not have f3 across partitions. In the later case, I/O wait time would be minimized, but f3 would create memory contention on the main memory and cause cache fetch interference to applications across all partitions.

While such side effects are simple facts that should be noticed by system developers, they are very difficult to recognize by conventional testing methods. Each partition of an IMA system is developed by separate entities and tested in isolation. During this phase of development, other hardware flows that exist for other applications in different partitions may be undefined and hence untestable. One practical way of performing early testing is to overload the I/O buses with imaginary data to consider the worst case, but this is too pessimistic. Thus, using a tool, such as ASIIST, that supports IMA is valuable during this phase of hardware I/O architecture design.

ASIIST also provides table and graphical views (Figure 5 and 6) for subsets of the model so that the user can easily make common changes to the model such as assigning applications to different IMA partitions or modifying partition sizes and changing I/O data paths. Each analysis focuses on different perspectives of the model. For schedulability analysis, ASIIST shows the processors, partitions, processes, and threads, where the user can drag and drop a process onto other partitions or processors. For bus delay analysis, ASIIST shows the hardware architecture of the system with data paths on top of the hardware components. The paths are colored by how close they are to their latency deadlines and can be filtered in various ways to see only what interests the user. A graphical user interface (GUI) allows the user to configure the settings for I/O and have the data path generated automatically. Users can easily change settings to test special features (e.g. PALS, Real-Time Bridges) that may be used for IMA systems to see their benefits and update other configurations to match the newly added features. More details about ASIIST support for each feature are explained in their corresponding sections.

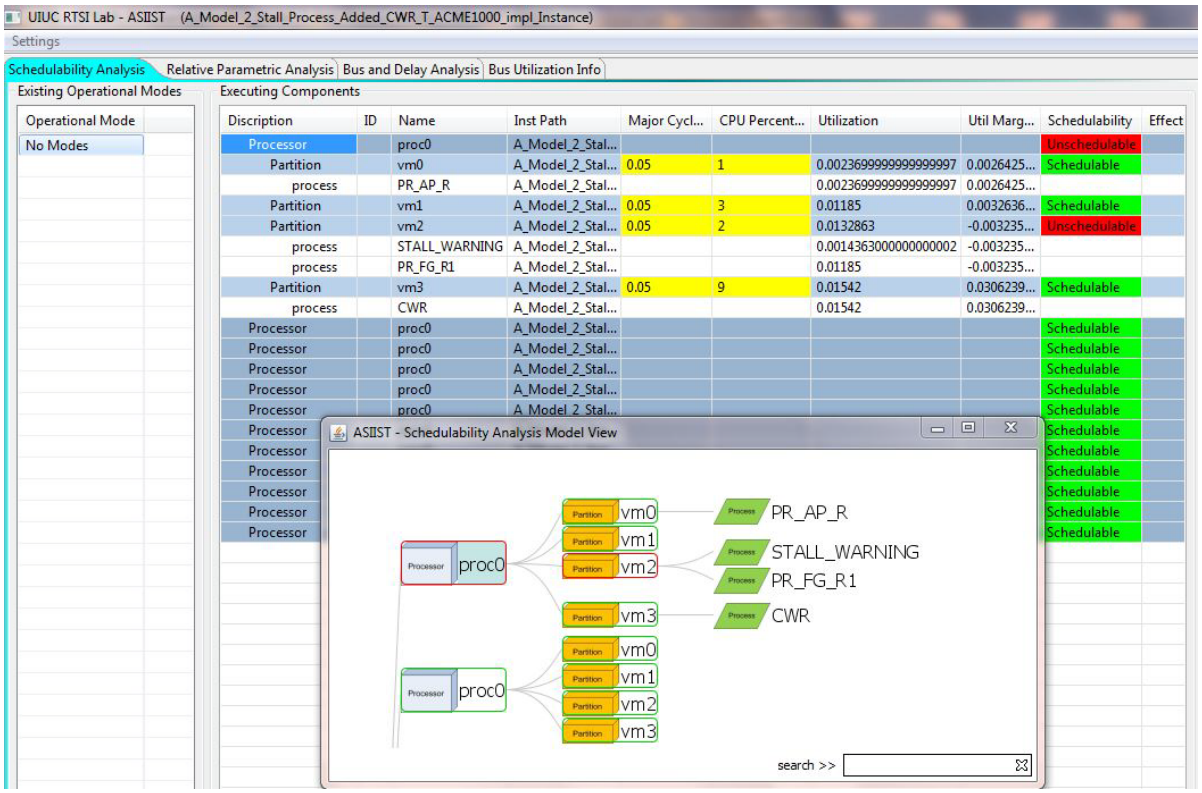

Figure 5. Schedulability Analysis Table and Graph View

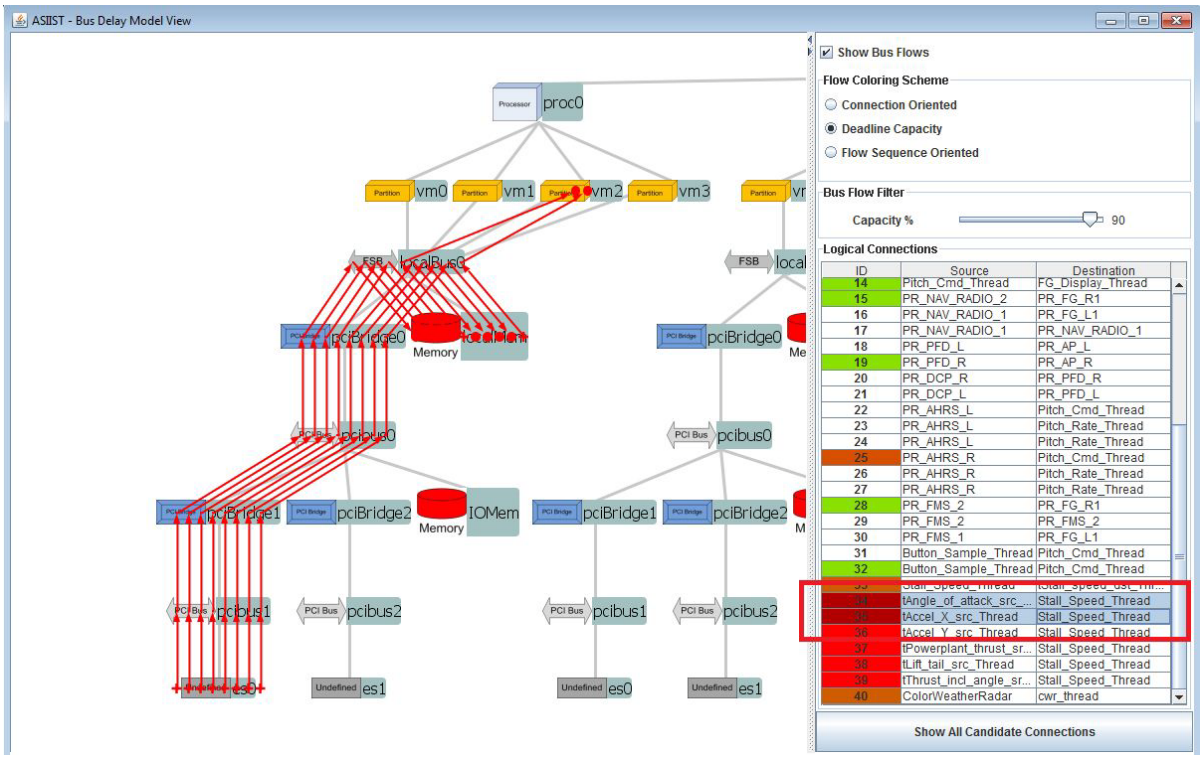

Figure 6. Bus Delay Analysis Graph View

I/O Management System for IMA

The framework introduced above can provide early analysis results based on high-level models, but the derivation of tight bounds is often compromised by the unpredictability inherent in COTS peripherals. As an example, consider a high-performance network card using DMA mode. The peripheral autonomously transfers packets in main memory as soon as it receives them. If packets suffer jitter on the network, the peripheral could potentially burst a large amount of data on the bus, resulting in significant delay for other I/O flows and increased memory contention for software tasks. This is particularly problematic if the exact arrival time of packets cannot be determined: a bursty I/O flow that must be processed by a software partition could potentially be transmitted while another, more critical partition is running on the CPU, resulting in undue timing interference.

In [5], we introduced a real-time I/O management system that enables us to efficiently perform peripheral traffic shaping. The management system enforces strict timing reservations for peripheral traffic, e.g. it bounds the maximum amount of data that a peripheral can send and/or receive in a given time period. In this way, the maximum traffic burstiness can be controlled, resulting in improved delay bounds. Furthermore, using our management system the designer can synchronize I/O traffic with the CPU schedule, prohibiting high-throughput peripherals from transmitting during critical software partitions.

System Components and Implementation

The simplification in system design comes at the cost of adding two new hardware components to the system, as shown in Figure 7. A real-time bridge is interposed between a peripheral and the rest of the system, providing the actuation mechanism to control peripheral bus accesses. The peripheral scheduler [1] enforces traffic timing reservations by controlling all real-time bridges. The prototype implementation described in [5] is based on FPGA devices, but a commercial implementation could potentially integrate both components in the motherboard chipset, with the addition of external RAM banks for buffering.

[1] The peripheral scheduler is called *reservation controller* in [5].

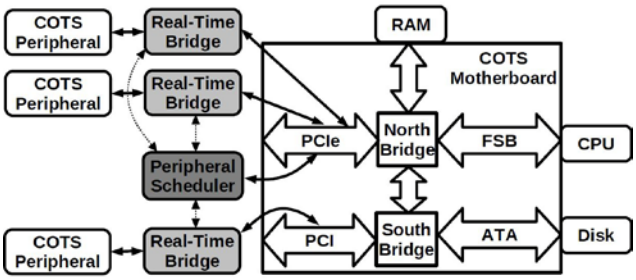

Figure 7. Real-Time I/O Management System

A block diagram for the implemented real-time bridge is shown in Figure 8, controlling an ethernet network peripheral (TEMAC in the figure). Incoming network packets are first buffered in a local bridge memory (FPGA DRAM). The *Bridge DMA Engine* then transmits the packets from local memory to main memory (host DRAM) when allowed to do so by the peripheral scheduler. In a similar way, outgoing packets are first transferred from main memory to local memory by the Bridge DMA Engine, and then sent through the controlled network peripheral.

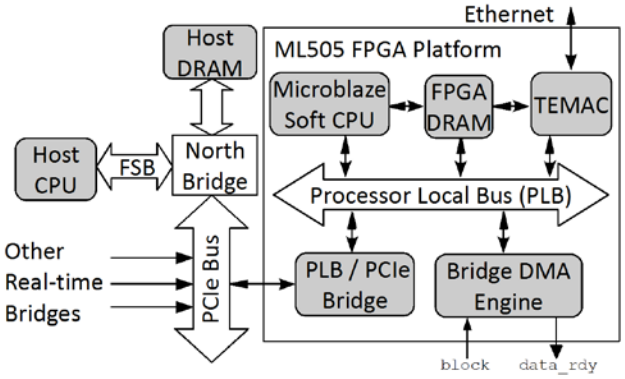

Figure 8. Real-Time Bridge Prototype

The operation of the peripheral and real-time bridge requires two software drivers. The *host driver* runs on the main CPU (host CPU), while the *FPGA driver* runs on a local CPU in the real-time bridge (a Microblaze soft core in our prototype). The host driver consists of a high-level driver, which creates the standard OS-level interface for a network card, and of a DMA interface communicating with the Bridge DMA Engine. The driver is completely transparent: from the application's perspective, there is no difference between using a network card directly or through a real-time bridge. Furthermore, the host driver can be reused for any network card.

The FPGA driver consists of a low-level driver, which handles the peripheral hardware, and of a corresponding DMA Interface. The DMA Interface is fully reusable. The low-level driver depends on the specific peripheral, but it can be adapted from available code. For example, in our prototype we were able to reuse the low-level portion of an existing Linux driver, with only limited modifications.

To simplify wiring, only two bits of information are exchanged between the peripheral scheduler and each real-time bridge: the *data_ready* signal informs the peripheral scheduler if there is buffered data to be transmitted on the real-time bridge, and the *block* signal is used to allow or prohibit the Bridge DMA Engine from transmitting data on the PCI-Express interconnection. Furthermore, in [17] we show how the peripheral scheduler can be synchronized with the CPU scheduler. In this way, the traffic enforcement policy can be changed based on the currently running software partition, for example, to only allow transmission of data flows required by the partition itself.

ASIIST Support

The use of real-time bridges is modeled in two ways. Users may add the physical representation of the bridges as AADL components and modify all the hardware data paths to go through the bridge. However, this may require modeling work when the user is searching for different options. Thus, we have defined some AADL properties that can be associated to each peripheral: a Boolean property of whether a real-time bridge is used, the transmission speed of the bridge, a transmission period and the maximum amount of data that can be sent or received during one period. Using the setting of the real-time bridge from the peripherals and bus tree architecture, ASIIST infers the set of flows that are allowed to contend in each bus segment. When the peripheral scheduler is synchronized with the CPU scheduler, real-time bridges hold lists of allowed IMA partitions which are used to derive the set of interfering flows, considering temporal partitions as well. The memory and bus analyses of previous sections are then applied based on the set of interfering flows. Finally, the analysis described in [5] enables the computation of precise bounds for the buffering delay and required buffer memory on each real-time bridge.

It is important to notice that a mixed solution is possible and supported by the tool, e.g. real-time bridges can be applied to only a subset of peripherals. An avionic computational node typically comprises a large number of peripherals, but only a few of them produce a large amount of data, in particular video that can significantly delay tasks and other data flows. Therefore, real-time bridges are only needed to control high-throughput peripherals.

Avionics Example

We describe a generic avionics example to demonstrate the various design decisions that can be made using ASIIST and show the analysis results. The model example we use is combining a Mission Control Computer (MCC) system with a pre-existing Flight Control System (FCS). We have an existing computing module which consists of a single core processor, main memory, local memory, and multiple peripherals connected through a PCI bus tree. Initially, FCS applications such as the Flight Guidance (FG) system and the Auto pilot (AP) system are assigned to one of the IMA partitions of the processor. A stall warning system application (SWSA) which is an optional feature of the FCS is added to the processor. Then we try to merge some of the features of the MCC system into more IMA partitions of the processor and demonstrate what may happen. Numerical data values used in the model are manually modified to demonstrate problematic situations for the purpose of illustration.

The MCC system consists of Radar Control, Targeting, Weapon control, Controls and Display functionalities. Each function is implemented by multiple processes. The processes we will consider include radar tracking, target designation, target tracking, weapon selection, weapon trajectory, and HUD (head-up display) Display.

Table 1 shows the list of applications and their nominal[2] initial worst case execution times which could be computed through timing analysis. Periodic processes have pre-determined period requirements, and some applications have latency requirements that give a deadline for how fast the output data must be delivered to an actuator. For example, a weapon

[2] Numbers shown in Table 1 are for illustration only and have no relation with any known systems. They are based on the technical report by http://www.sei.cmu.edu/reports/90tr008.pdf

selection's response must occur within 200 ms to provide the aircrew with an appearance of instantaneous response. Currently SWSA is not implemented and thus we do not have its WCET. WCET for MCC applications are assumed to have been tested in isolation from other applications. Figure 9 describes the hardware architecture of a computing system where the applications reside. Network data traffic arrives through one of the end system devices.

Table 1. Application Parameters

Application	WCET (init)	Period	Partition
AP	0.0237 ms	0.01 sec	vm0
FG	1.185 ms	0.1 sec	vm1
SWSA	N/A	0.01 sec	vm2
RadarC	2 ms	0.04 sec	vm3
TargetDesig	1 ms	0.2 sec	vm4
TargetTrack	4 ms	0.04 sec	vm4
WeapSel	1 ms	0.1 sec	vm5
WeapTraj	7 ms	0.1 sec	vm5
HUDApp	6 ms	0.052 sec	vm6

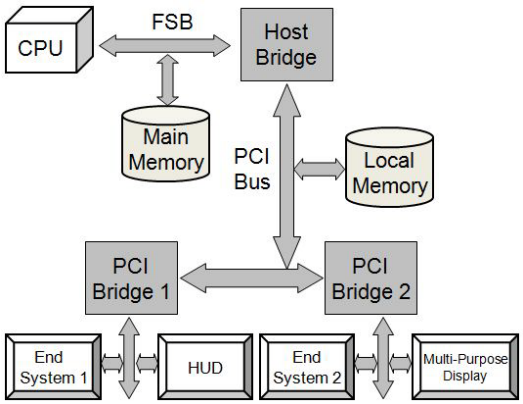

Figure 9. Hardware Architecture

The HUD and Multi-Purpose Display (MPD) are display devices where data is sent to be seen by the pilot. Data sent to the HUD and MPD must be directly written to their registers. The CPU is partitioned for IMA and the local memory is a large but slow memory where data may be saved for later access.

Demonstration with Example

Initially, we imagine an IMA system with only the AP and FG assigned to its partitions. We assume that we do not have the source code for the SWSA which we wish to add. We use a Simulink model to generate a preliminary SWSA code that will be analyzed by the TA to make an estimate of its WCET. Using this value, we add the SWSA model and check its schedulability.

As shown in Figure 10, the partition size is adjusted through ASIIST to find the values 1%, 3%, 2%, for partitions vm0, vm1, and vm2, respectively. The initial WCET of SWSA was estimated to be 0.014363 ms. After running the memory contention analysis, we notice that vm1 becomes unschedulable due to the FG's increased execution time (Figure 11). We increase the partition size of vm1 to 4% to make it schedulable.

Now we add the applications of the MCC to the partitions shown in Table 1. Initially, the partitions are schedulable with the partition sizes of 10%, 20%, 16%, and 22% for partitions vm3, vm4, vm5, and vm6, respectively. However, after running the bus delay analysis and adding the I/O wait states to the WCET, the partitions are no longer schedulable.

Executing Components

Description	Name	CPU Perce...	Utilization	Schedulability	Initial Exec time(sec)	Period(sec)
Processor	proc0			Schedulable		
Partition	vm0	1	0.00237	Schedulable		
process	PR_AP_R		0.00237			
thread	Pitch_Rate_Thread		0.00237		2.37E-5	0.01
Partition	vm1	3	0.0118500...	Schedulable		
process	PR_FG_R1		0.0118500...			
thread	Pitch_Cmd_Thread		0.0118500...		0.001185	0.1
Partition	vm2	2	0.0014363...	Schedulable		
process	STALL_WARNING		0.0014363...			
thread	Stall_Speed_Thread		0.0014363...		1.4363000000000001E-5	0.01

Figure 10. Schedulability Result for Adding SWSA

Executing Components

Description	ID	Name	CPU Perce...	Utilization	Schedulability	Effective Exe...	Initial Exec time(sec)	Period(sec)
Processor		proc0			Unschedulable			
Partition		vm0	1	0.00290333...	Schedulable			
process		PR_AP_R		0.00290333...				
thread		Pitch_Rate_Thread		0.00290333...		0.0000290333...	2.37E-5	0.01
Partition		vm1	3	0.01787666...	Unschedulable			
process		PR_FG_R1		0.01787666...				
thread		Pitch_Cmd_Thread		0.01787666...		0.0017876666...	0.001185	0.1
Partition		vm2	2	0.00432696...	Schedulable			
process		STALL_WARNING		0.00432696...				
thread		Stall_Speed_Thread		0.00432696...		0.0000432696...	1.4363000000000001...	0.01

Figure 11. After Applying Memory Contention Analysis

The partition sizes of vm3, vm4, vm5, and vm6 must be increased to 15%, 28%, 20%, and 33% respectively (Figure 12). In this case, the total sum of all the partition sizes is 103%, which is not realizable. The increase is essentially due to the large volume of data that must be written to the HUD and MPD.

Partition	vm1	4	0.017897...	Schedulable
process	PR_FG_R1		0.017897...	
Partition	vm2	2	0.005063...	Schedulable
Partition	vm3	15	0.074165...	Schedulable
process	PR_RadarControl		0.074165...	
Partition	vm4	28	0.150070...	Schedulable
process	PR_TargetDesignation		0.015938...	
process	PR_TargetTracking		0.134131...	
Partition	vm5	20	0.102879...	Schedulable
process	PR_WeaponSelection		0.019076...	
process	PR_WeaponTrajectory		0.083802...	
Partition	vm6	33	0.176318...	Schedulable
process	PR_HUD		0.176318...	

Figure 12. MCC Assignment

We will now try using a Real-Time bridge to reduce the size of the partition. The RT bridge feature is used on end system 1 to buffer any data that is received through end system 1 and allow it to be sent only during partitions vm3, vm4, and vm5. The RT bridge is set to send data at a period of 100 ms with a buffer size of 500 B. This actually removes memory contention in vm1 so that it can remain with the size of 3%. Basically vm6 gets the most benefit due to the volume of data that must be sent directly to the HUD by the HUD application. Since the HUD shares a PCI bus with end system 1, the delay caused by this shared bus segment is reduced (Figure 13).

Executing Components

Description	ID	Name	CPU Perce...	Utilization	Schedulability	Effective Exe...	Initial Exec time(sec)	Period(sec)
Processor		proc0			Schedulable			
Partition		vm0	1	0.002373...	Schedulable			
Partition		vm1	3	0.011851...	Schedulable			
Partition		vm2	2	0.001857...	Schedulable			
Partition		vm3	15	0.075783...	Schedulable			
process		PR_RadarControl		0.075783...				
thread		RadarCTRL_THREAD		0.075783...		0.0030313407...	0.0020	0.04
Partition		vm4	29	0.152243...	Schedulable			
process		PR_TargetDesignation		0.016493...				
thread		TargetDesig_THREAD		0.016493...		0.0032987531...	0.0010	0.2
process		PR_TargetTracking		0.135750...				
thread		TargetTrack_THREAD		0.135750...		0.0054300011...	0.0040	0.04
Partition		vm5	20	0.104913...	Schedulable			
process		PR_WeaponSelection		0.019908...				
thread		WeapSel_THREAD		0.019908...		0.0019908874...	0.0010	0.1
process		PR_WeaponTrajectory		0.085004...				
thread		WeapTraj_THREAD		0.085004...		0.0085004559...	0.0070	0.1
Partition		vm6	28	0.146382...	Schedulable			
process		PR_HUD		0.146382...				
thread		HUD_THREAD		0.146382...		0.0076118850...	0.0060	0.05200000...

Figure 13. Final Result of IMA Partition Assignment

Related Work

There are several real-time analysis tool sets that support AADL and that are similar to ASIIST. Ocarina [18] is a tool set which allows AADL model manipulation, generates formal models, performs scheduling analysis and generates distributed applications.

The Furness toolset [19] presents a translation of AADL models into the real-time process algebra ACSR (Algebra of Communicating Shared Resources) that allows schedulability analysis of AADL models as well as an AADL simulator to visually track system utilization. However, none of the above tools focuses on designing the architecture of the bus.

SymTA/S [20] is a tool that has been developed for system-level performance of real-time properties. However, it does not use explicit abstractions of bus protocols mainly because it is targeted for system-on-chip which uses heterogeneous scheduling techniques. The authors of [21] share our point of view for using model-based integration tools to build system architecture.

Summary and Conclusion

Managing the effects of architectural decisions throughout the design and development process is essential for controlling the cost and schedule of complex system development projects. We have presented a set of analysis techniques to support decision making at the system architectural level. We have instantiated these analysis approaches in a tool chain that provides automated decision support. Use of this tool chain greatly simplifies the tracking and management of the effects of subsystem and component changes in a comprehensive quantitative manner.

There are several directions for future work. Topics of interest include obtaining more accurate hardware models and exploring the effects of data caching and instruction pipelining. As models of new processor architectures become available, we can incorporate them into the timing analysis portion of the tool chain. We can calibrate the results of the analyses we obtain from the tool chain to reflect the relative efficiency of developed code versus auto-generated code. We can also explore the use of analysis to derive processor requirements for meeting a set of software and communication requirements, rather than starting from an assumed processor architecture to determine a yes-or-no answer to whether these requirements are met. Finally, we can explore adding support for additional complexity-reducing design patterns beyond PALS to simplify the development of complex systems.

References

[1] Sha L., 2003, Real-time virtual machines for avionics software porting and development, Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 123–135.

[2] Mohan, S., M.-Y. Nam, R. Pellizzoni, L. Sha, R. Bradford, and S. Fliginger, Dec 2009, Rapid Early-Phase Virtual Integration, Proceedings of IEEE Real-Time Systems Symposium (RTSS'09), pp. 33–44.

[3] Pellizzoni, R., B. Bui, M. Caccamo, and L. Sha, Dec 2008, Coscheduling of CPU and I/O transactions in COTS-based embedded systems, Proceedings of IEEE Real-Time Systems Symposium (RTSS), pp. 221–231.

[4] Le Boudec, J.-Y. and P. Thiran, 2001, Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Springer.

[5] Bak, S., E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha, Dec 2009, Real-time control of I/O COTS peripherals for embedded systems, Proceedings of IEEE Real-Time Systems Symposium (RTSS'09), pp.193-203.

[6] Al-Nayeem, A., M. Sun, X. Qiu, L. Sha, S. P. Miller, D. D. Cofer, Dec 2009, A Formal Architecture Pattern for Real-Time Distributed Systems, Proceedings of IEEE Real-Time Systems Symposium (RTSS'09), pp.161-170.

[7] Miller, S., D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem, 2009, Implementing Logical Synchrony in Integrated Modular Avionics, in Proceedings of the 28th Digital Avionics Systems Conference (DASC 2009), pp. 1.A.3-1-1.A.3-12.

[8] Healy, C. A., D. B. Whalley, and M. G. Harmon, Dec. 1995, Integrating the timing analysis of pipelining and instruction caching, Proceedings of IEEE Real-Time Systems Symposium (RTSS), pp.288-297.

[9] Mohan, S., F. Mueller, D. Whalley, and C. Healy, Mar. 2005, Timing analysis for sensor network nodes of the atmega processor family, Proceedings of IEEE Real-Time Embedded Technology and Applications Symposium (RTAS), pp.405-414.

[10] White R., Apr. 1997, Bounding Worst-Case Data Cache Performance. PhD thesis, Dept. of Computer Science, Florida State University.

[11] Cadar, C., D. Dunbar, and D. Engler, 2008, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, In the USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[12] Pellizzoni, R., J.-J. Schranzhofer, M. Chen, M. Caccamo, and L. Thiele, Mar. 2010, Worst case delay analysis for memory interference in multicore systems, Proceedings of Design, Automation and Test in Europe (DATE).

[13] Pellizzoni, R., and M. Caccamo, Mar. 2010, Impact of peripheral processor interference on wcet analysis of real-time embedded systems. IEEE Trans. on Computers, pp. 400-415.

[14] PCI-SIG. http://www.pcisig.com/specifications.

[15] Nam, M.-Y., R. Pellizzoni, L. Sha, and R. M. Bradford, June 2009, ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs, Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp.11-22.

[16] Feiler, P., B. Lewis, and S. Vestal, Oct. 2006, The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems, Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, pp.1206-1211.

[17] Pellizzoni, R., 2010, Predictable and Monitored Execution for COTS-based Real-Time Embedded Systems. PhD thesis, University of Illinois at Urbana-Champaign.

[18] Hugues, J., B. Zalila, L. Pautet, and F. Kordon, 2008, From the prototype to the final embedded system using the Ocarina AADL tool suite, ACM Trans. on Embedded Computing Sys, pp. 1-25.

[19] Sokolsky, O., I. Lee, and D. Clarke, 2009 Process-algebraic interpretation of AADL models, Proceedings of the 14th Ada Europe International Conference on Reliable Software Technologies, pp. 222-236.

[20] Henia, R., A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, Mar 2005, System level performance analysis – the SymTA/S approach, IEE Proceedings Computers & Digital Techniques, pp 148-166.

[21] Porter, J., G. Karsai, P. Volgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits, 2009, Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation, Proceedings ACES-MB Workshop (MoDELS'08), pp. 20-34

Acknowledgements

This work is supported in part by a grant from Rockwell Collins, by a grant from Lockheed Martin, by NSF CNS 06-49885 SGER, NSF CCR 03-25716 and by ONR N00014-05-0739. Opinions, findings, conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors. All numeric values used are representative of generic avionics applications and have no connection to any existing, or future, Rockwell Collins or Lockheed Martin products.