Modeling Towards Incremental Early Analyzability of Networked Avionics Systems Using Virtual Integration

MIN-YOUNG NAM, University of Illinois at Urbana-Champaign
KYUNGTAE KANG, Hanyang University
RODOLFO PELLIZZONI, University of Waterloo
KYUNG-JOON PARK, Daegu Gyeongbuk Institute of Science & Technology
JUNG-EUN KIM and LUI SHA, University of Illinois at Urbana-Champaign

With the advance of hardware technology, more features are incrementally added to already existing networked systems. Avionics has a stronger tendency to use preexisting applications due to its complexity and scale. As resource sharing becomes intense among the network and the computing modules, it has become a difficult task for the system designer to make confident architectural decisions even for incremental changes. Providing a tailored environment to model and analyze incremental changes requires a combination of software tools and hardware support. We have built a virtual integration tool called ASIIST which can provide a worst-case end-to-end latency of data that is sent through a network and the internal bus architecture of the end-systems. Also, we have devised a new real-time switching algorithm which guarantees the worst-case network delay of preexisting network traffic under feasible conditions. With the real-time switch support, ASIIST can provide an early modularized analysis of the end-to-end latency to make architectural design choices and incremental changes easier for the user.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; B.2.2 [**Arithmetic and Logic Structures**]: Performance Analysis and Design Aids—*Worst-case analysis*

General Terms: Design

Additional Key Words and Phrases: AADL, avionics system design, end-to-end delay analysis, modeling, switching algorithm, virtual integration

ACM Reference Format:
Nam, M.-Y., Kang, K., Pellizzoni, R., Park, K.-J., Kim, J.-E., and Sha, L. 2012. Modeling towards incremental early analyzability of networked avionics systems using virtual integration. ACM Trans. Embedd. Comput. Syst. 11, 4, Article 81 (December 2012), 23 pages.
DOI = 10.1145/2362336.2362348 http://doi.acm.org/10.1145/2362336.2362348

This work was supported in part by NSF CNS 06-49885 SGER, NSF CCR-3-25716, and by ONR N00014-05-0739. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of sponsors.
Authors' addresses: M.-Y. Nam, Department of Computer Science, University of Illinois at Urbana-Champaign; K. Kang (corresponding author), Department of Computer Science and Engineering, Hanyang University; email: ktkang@hanyang.ac.kr; R. Pellizzoni, Department of Electrical and Computer Engineering, University of Waterloo; K.-J. Park, Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology; J.-E. Kim, and L. Sha, Department of Computer Science, University of Illinois at Urbana-Champaign.

DOI 10.1145/2362336.2362348 http://doi.acm.org/10.1145/2362336.2362348

1. INTRODUCTION

In many hard real-time avionics systems, more and more features are being added to faster but less expensive hardware. Thus, hardware resources such as computation and network bandwidth are increasingly being shared by multiple applications, leading to rapid increases in the size and complexity of the overall systems. Avionics systems are now so large and complicated that a new system is rarely designed from scratch. Normally, new features are added on top of an already verified system or network of systems, with the aim of controlling the cost of the project. However, even with this effort, if the additional new features have side-effects to other applications due to shared resources or network connections, they would all have to be reanalyzed for temporal properties such as schedulability or end-to-end latency. This requirement is difficult to reconcile with the early decisions on application assignments and system architecture design that a system designer must make. These early system-level design decisions are very costly to change later, due to their ramifications across systems. Still, designers are making these decisions based on their past experience and simplified numeric properties of hardware (CPU speed, memory size, bus speed, etc.).

Fortunately, a better approach is now becoming available, in the form of *virtual integration* which is being adopted by the avionics industry as well as other manufacturers of Cyber-Physical Systems (CPS), which include the automotive and medical industries. Avionics systems also have to meet many safety-critical criteria, which are impossible to verify manually when a system is very complicated. Virtual integration [Mohan et al. 2009] is a concept of performing all the required analysis and estimation before actual integration using software and hardware models of a system. Virtual integration helps system designers to test multiple options, make modifications, and check safety-critical requirements relatively quickly. The SAE AADL (Architecture Analysis and Design Language) [SAE 2009] is an architecture description language which has become popular for virtual integration of avionics systems. Using AADL, we can define models of software and hardware components, construct a virtual system, and build tools to analyze them.

We will present a case study that demonstrates the virtual integration of an avionics system, in which an environmental monitoring feature is being added to a functioning network of systems. Figure 1 is a diagram of an aircraft with this added feature for environmental monitoring, which consists of several visual sensors which allow a pilot to look in different directions without turning the aircraft. This feature is found on military aircrafts where identifying and monitoring the surrounding for enemy or friendly presence can assist the mission whether it is a reconnaissance mission or a search and rescue mission. In designing such systems, many questions must be answered, for instance, how many sensors can the network support in real time without damaging preexisting data-flows, or what is a feasible network topology. Such questions need to be answered prior to physical integration, when the discovery of scheduling problems will lead to expensive changes.

We show how our real-time monitoring application might be introduced without violating any delay bounds. To do this, we combine our Architecture-Specific I/O Integration Support Tool (ASIIST) with a new design of a real-time switch. ASIIST can analyze the worst-case delay of data traffic within a computational module for an arbitrary bus-tree architecture specified in AADL. ASIIST considers the effect of multiple applications running on a single computational module that shares the internal bus. The additional network delay imposed by the transfer of video frames across multiple switches and their connections in our application is determined by the switching algorithm.

Most previous network switch designs have been focused on best-effort traffic, rather than on the provision of any guarantee of latency. Thus, when new traffic is introduced, the systems that already share the network are affected in a nondeterministic way.

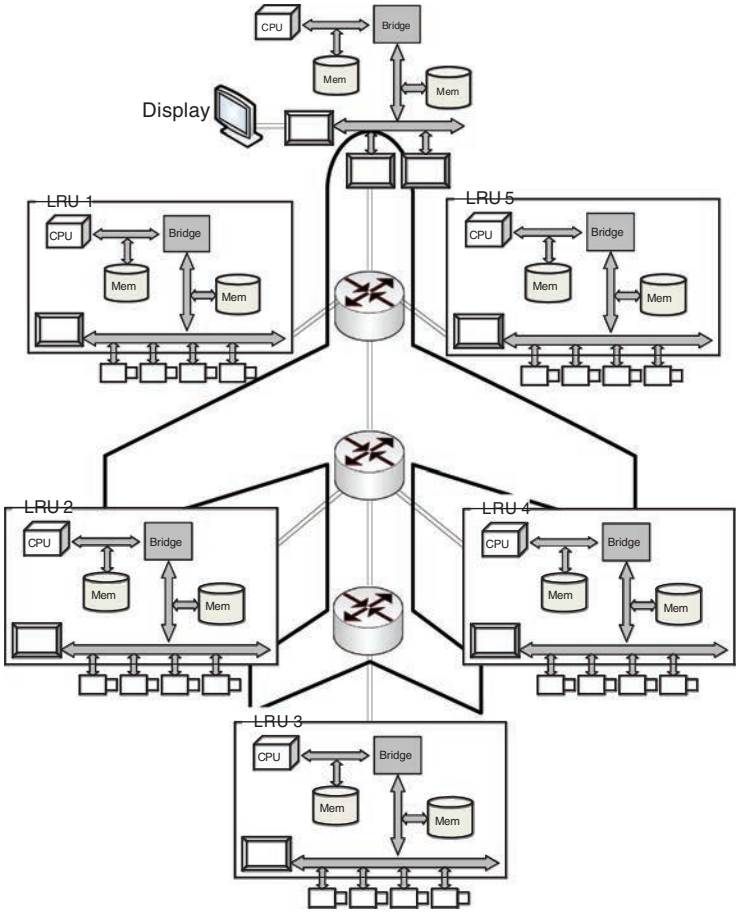


Fig. 1. Network system overview.

This means that all the systems with any connection to the network must be retested, even those that have not been changed. We have designed a real-time switching algorithm which is more friendly to incremental changes, because its delay is guaranteed, provided that certain bounding requirements are met. By combining this new switch with the virtual integration capability of ASIIST, we can perform *early modularized predictive analysis* of the feasibility of adding a new *real-time networking* feature, such as the environmental monitoring application described before. Our ultimate aim is to reduce integration cost and guarantee end-to-end latency.

The main contribution of this article is the combination of virtual integration of computational nodes with a new switching algorithm to guarantee the end-to-end latency, which is the sum of delays within nodes and across the network. We use virtual integration to realize the sharing of an internal bus among multiple applications within an Integrated Modular Avionics (IMA) [Aeronautical Radio Inc. 1991] environment. The new switching algorithm reduces the effect of new network features on the existing data network traffic, by guaranteeing the latency of preexisting network tasks. Thus, incremental development becomes easier for the user to understand.

The remainder of this article is organized as follows: In Section 2, we explain other related work. Section 3 explains our case study example of adding a new networking

feature. Section 4 explains how we model and analyze the worst-case bus delay of end-systems by using AADL and our tool ASIIST. Section 5 introduces a new real-time switching algorithm which can provide real-time guarantees for the data network and can assist in incremental development. In Section 6, we combine the previous analyses to get a true worst-case end-to-end latency bound. Section 7 gives a demonstration of analyzing the network system and comparing design choices of different internal bus architectures and the number of cameras to install. Finally, Section 8 concludes.

2. RELATED WORK

The concept of IMA has been around for more than ten years. However, as the avionics companies try to benefit from using more Commercial Off-The-Shelf (COTS) components to reduce overall costs and accelerate system integration, building a deterministic and reliable IMA system becomes more challenging. Previous IMA system implementations used the ARINC (Aeronautical Radio, Incorporated) 659, commercially known as SAFEBus [Hoyme and Driscoll 1992], as a backplane data bus. SAFEBus is a fault-tolerant time-division multiplexed bus which requires Bus Interface Units (BIUs) that are synchronized to handle bus accesses. This configuration has been successfully used in the Boeing 777 integrated Airplane Information Management System (AIMS) [Driscoll and Hoyme 1992]. Previous analyses have assumed that appropriate specialized I/O techniques [Audsley and Wellings 1996] and the tools needed to support IMA [Lee et al. 2000] are protected from the side-effects of using COTS components by the SAFEBus.

The most widely used standard for Aircraft Data Networks (ADNs) has been ARINC 429, which supports up to 100kbps. ARINC 629 was successfully applied to the Boeing 777 with a data rate of 2Mbps, but this required customized hardware. As COTS hardware has become more advanced, the avionics companies have an increasing interest in trying to use more COTS components to reduce costs [Baker 2002; Reynolds 1998]. New standards, such as ARINC 664 [Aeronautical Radio Inc. 2005] (a star-topology switched Ethernet network based on IEEE 802.3), follow this trend by making wider use of COTS components [Schuster and Verma 2008]. Legacy ARINC 429 network components can be mapped to ARINC 664 networks through gateways and routers. Methods of estimating end-to-end delays in ARINC 664 networks are receiving increasing attention [Scharbarg et al. 2009].

Several other real-time analysis tool-sets are similar to ASIIST, and also support AADL. Cheddar [Singhoff et al. 2005] is a set of tools that performs scheduling simulations and feasibility tests for quick prototyping of real-time schedulers. Ocarina [Hugues et al. 2008] is a tool-set which permits the manipulation of AADL models, generates formal models, performs scheduling analysis, and generates distributed applications. The Furness tool-set [Sokolsky et al. 2009] translates AADL models into the real-time process algebra ACSR (Algebra of Communicating Shared Resources), which allows schedulability analysis of AADL models. Furness also provides an AADL simulator to allow system utilization to be tracked visually. However, none of the aforesaid tools is focused on the way in which architecture of the internal bus is combined with the network. Thus, it is very difficult for users of these tools to change bus topology using play-compatible components of a hardware model; and this makes it more difficult to understand the architectural properties of buses.

SymTA/S [Henia et al. 2005] is a tool that has been developed to model the system-level performance of real-time tasks. However, it does not use explicit abstractions of bus protocols, mainly because SymTA/S is targeted at system-on-chip designs, which use heterogenous scheduling techniques. Porter et al. [2009] share our approach to the use of model-based integration tools to build system architectures. While suggesting a

model exchange with AADL, they use a language called Embedded Systems Modeling Language (ESMoL) for simpler graphical entry and focus on Model of Computation (MoC) and integration with legacy modeling and embedded systems tools.

There has been extensive research on the design of network switches [Weller and Hajek 1997; McKeown 1999; Neely et al. 2007; Gupta et al. 2009], but most of it has been focused on the development of efficient switching algorithms, within a graph-theoretic framework, in order to realize a stability region suitable for general traffic. An asymptotic logarithmic delay bound can be derived from the stability condition [Neely et al. 2007]. A more recent switching policy [Gupta et al. 2009] has been shown to be clearance-time-optimal as well as throughput-optimal (and this policy is incorporated into our framework). However, very few of these studies have explicitly considered real-time applications.

Research on network infrastructure for real-time applications has usually focused on specific, small-scale network architectures. For example, prioritized bus and ring networks have been used in small real-time systems [Sha et al. 1990; Gopalakrishnan et al. 2004], but they are not designed for high-speed network backbones, such as those of WANs. Rexford et al. [1998] proposed a router for real-time communication, but this was designed to support deadline-based scheduling, which requires significantly different hardware; nor is this router designed for the backbones of high-speed networks. The same applies to the real-time Ethernet switch proposed by Venkatramani and Chiueh [1997].

An efficient switch design for real-time applications has recently been proposed by Qixin et al. [Wang et al. 2008], who provides a mechanism for guaranteeing the allocation of a certain number of communication slots to a task over a fixed time interval, under the assumption of deterministic and periodic real-time traffic. This work provided us with some inspiration but it is significantly different because we are able to guarantee a delay bound for any feasible traffic, including nonperiodic traffic.

To the best of our knowledge, none of the work reviewed earlier involves virtual integration of real-time switch support, which is necessary to analyze the true end-to-end latency of networked COTS-based IMA systems. In this way, we incorporate the variable datapaths within end-systems and the shared network topology. The key benefit of our real-time switch arises from our ability to configure and evaluate the network's extensibility so that added features will not affect the guaranteed delay bounds of preexisting network connections for incremental changes.

3. CASE STUDY: REAL-TIME NETWORKING FEATURE

Redundant hardware components are necessary to protect safe-critical avionics systems from hardware failures. While this is obviously essential, it has disadvantages in terms of aircraft weight and maintenance costs. Thus, the elimination of any system also eliminates many redundant components, and this can save a lot of cost that is entailed during the long lifetime of an aircraft. Thus, utilizing the minimum amount of hardware, or being able to use existing hardware to support new features, is extremely important. We advance this goal by guaranteeing real-time properties, such as the end-to-end latency, of data connections through a network when a new networking feature is added to existing data traffic.

We now present a case study to demonstrate the importance and challenge of our work. Consider a scenario shown in Figure 1, which shows the conceptual design of an environmental monitoring system. Visual sensors (cameras) are positioned to capture still images or video from many angles, which provides the capability of looking in many directions without changing the direction of the aircraft. With this capability the aircraft can fly in a straight line while scanning all of the surroundings using image processing techniques. Only scanning in one direction is not effective when the speed

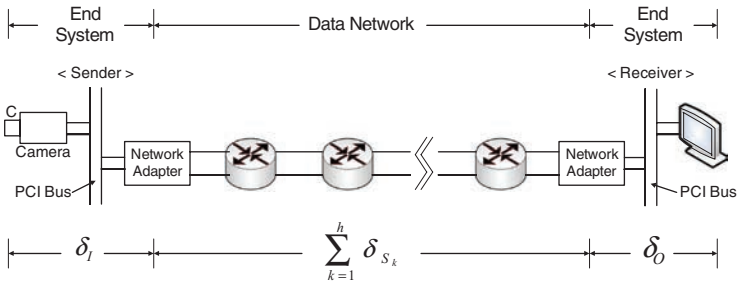


Fig. 2. Datapath in a network system.

of the aircraft is high. Sensing in all directions would greatly improve the efficiency of Search And Rescue (SAR) or reconnaissance missions in particular. The visual sensors are intended to be connected to an existing switching data network, which relays image data from the cameras to a designated monitoring device.

This system can naturally be partitioned by topology into end-systems and a data network. An end-system consists of the circuits connecting each camera to the network adapter on the sender side (the sending end-system) and the computational node connecting the network adapter to each monitoring device on the receiver side (receiving end-system). The receiving end-system not only forwards the image data, but also has to execute multiple applications, to maximize hardware utilization. The image data has to be preprocessed so that annotations and alerts can be provided to the pilot in real time. The receiving end-system also acts as common computational module which performs additional general tasks. In our model, the receiving end-system is running additional applications, including the Automatic Flight Control System (FCS), Color Weather Radar (CWR) system for visualizing weather satellite images, and an Inertial Navigation System (INS)/Global Positioning System (GPS)/Doppler navigation system that uses motion sensor (accelerometers, gyroscopes) and satellite data. Safety-critical hard real-time tasks, such as the FCS and INS/GPS/Doppler navigation system, which are directly related to the aircraft's safety are included. Other tasks are focused on visual display data because they constitute most of the heavy data traffic in modern avionics systems. In a IMA system, these tasks should safely share the resources of a computer system. Although these additional applications receive data from other sensors of the aircraft, they are omitted to simplify Figure 1.

Wired cables connect the network adapter on the sender side to the network adapter on the receiver side through several switches in the data network. Figure 2 provides a view of this structure. A system designer needs to know whether the existing architecture of end-systems and the data network are able to support the new networking feature meaning. Thus the designer has to find out whether the end-to-end latency of all the applications satisfies their several requirements.

Figure 2 shows the three sources of delay. Let δ_I denote the worst-case delay introduced between a camera and a network adapter, in the sending end-system; let δ_O be the worst-case delay introduced by the internal bus communication from the network adapter to the pilot's monitor; and let δ_{S_k} be the worst-case switching delay introduced by the kth switch in the route from a network adapter on the sender side to another network adapter on the receiver side. Since the propagation delay between the switches and the end-systems are negligible, the total worst-case delay between the network adapters of the sending and receiving end-systems can be simplified to $\sum_{k=1}^{h} \delta_{S_k}$, where h denotes the hop count in the data network, which is the number of switches traversed by each packet. Using this notation, we can now formulate $\bar{\delta}$, which is the worst-case

bound on the end-to-end latency.

$$\bar{\delta} = \delta_I + \sum_{k=1}^{h} \delta_{S_k} + \delta_O \tag{1}$$

We will first start with the analysis of the end-systems. The end-systems are composed of processors, storage elements, and peripheral I/O devices, using PCI-based COTS components for communication. As the hardware become more advanced, these end-systems now perform multiple tasks and the internal bus can be easily overwhelmed by the large data it receives. Although the bus capacity is satisfied, it does not necessarily mean that the bus delay remains within its requirements. In order to analyze the delay of bus communication in these complex end-systems, we have built a software tool named ASIIST (Application-Specific I/O Integration Support Tool) which is explained in the following section.

4. END-SYSTEM MODEL AND DESIGN

The numerous decisions to be made during the design of complex hard real-time avionics systems include the design of the PCI [PCI 2012] bus architecture using COTS components, the trade-off between computation time and bandwidth, and the use of DMA-like transactions. As the avionics industry moves towards the use of Integrated Modular Avionics (IMA) [Aeronautical Radio Inc. 1991], more decisions, such as the assignment of software applications to processor partitions, partition sizing, and partition order, must be made for the end-systems of a network.

IMA is an architectural concept for hosting multiple software applications on a networked set of computational nodes. Each node may run several software applications, which are temporally and spatially isolated for safety. This has been considered an attractive way of running COTS systems in terms of overall cost [Baker 2002], and major avionics companies are trying to implement IMA on COTS-based systems. In the Boeing 777, specialized backplane data bus, commercially known as SAFEBus [Hoyme and Driscoll 1992], was used to handle the unpredictable I/O traffic [Driscoll and Hoyme 1992]. However, modern applications require larger amounts of visual data, and the SAFEBus (60Mbps) is no longer fast enough. It would also be inadequate for our example environmental monitoring system. The way in which IMA-based systems share resources has also made the end-systems less predictable. If tens of applications share a computational node, it is difficult to estimate the bus delay within an end-system built with COTS components, even though the higher speed of such components is obviously beneficial.

To determine the worst-case delays in real-time applications at an early stage of design, we have built a tool name ASIIST. This is a provisional tool that is focused on the analysis of I/O bus delays for protocol-specific data-flows in hardware. When adding new applications, it is important to find out whether the bus throughput is adequate to support the new data traffic. In many cases, it is impossible to run all the applications that share common buses, until the final system has been coded, by which time it is too late to make the required changes without starting again. ASIIST is a virtual integration tool which uses AADL models and additional data-flow information to estimate bus delay within a computing system. Its aim is to provide designers with the information needed to allow them to try different bus architectures and I/O configurations without the need to implement whole systems.

4.1. Modeling of Bus Communication Using AADL

We will give a brief overview of AADL in this section and show how it is used, and how it can be extended to specify bus communication, using PCI as an example. This includes

the modeling blocks for the hardware architecture, specification of logical data-flows, PCI-protocol-specific hardware flows, and I/O configuration options.

4.1.1. Overview of AADL. AADL is a language based on 15 years of research, including the MetaH language developed by Honeywell Labs and several DARPA programs [Binns et al. 1996], and it has been applied to many industry examples. Currently its development is led by Peter Feiler at the Software Engineering Institute (SEI). AADL provides means of specifying the hardware and the software architecture of embedded systems. With AADL, we are able to perform various kinds of analyses, such as safety analysis, fault tolerance, schedulability, system latency, etc. It provides textual and graphical interfaces to allow users to build the architecture of a system composed of "components" and to specify the interaction among the components. The architecture also supports the development of tools that perform dynamic or static analysis based on the system specification.

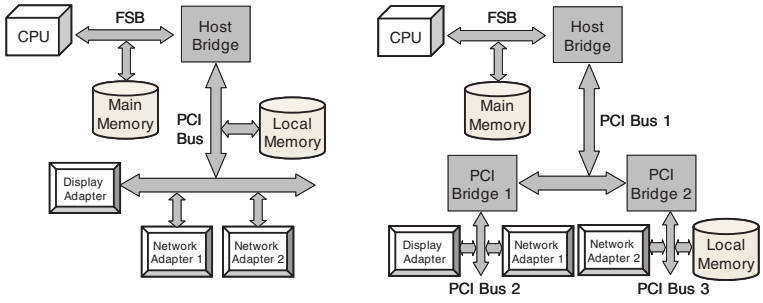
AADL components are divided into three categories: software, execution platform, and system. Software components include `process`, `thread`, `thread group`, `subprogram` and `data` components. To represent the infrastructure, we have the `processor`, `memory`, `bus` and `device` as execution platform components. The `system` component is a special composite component which is used to divide components into groups or encapsulate components to distinguish them from others as a separate system object. It could also be used to substitute any new abstract entity that is not predefined before. Software components are bound to execution platform components to represent where it is executed, for `threads`, or where it resides, for `data` components.

Property specification plays a major role in AADL since properties add extra information that cannot be expressed by structural descriptions. The core AADL has standard predeclared properties that support real-time scheduling as well as other areas of research. All bindings are done by property associations in AADL. AADL also provides the syntax to add new user-defined properties. AADL is also extensible so that we can add sublanguages as annexes to describe more complicated semantics that can be processed as part of the specification of an AADL model.

4.1.2. Modeling Block Definition for PCI Bus Architecture. The Peripheral Component Interconnect (PCI) is the current standard family of communication architectures for motherboard-peripheral interconnection in personal computers; it is also widely popular for embedded systems [PCI 2012]. While ASIIST can perform the worst-case analysis of delay in any bus protocol, it is beneficial to achieve tighter bounds on delays in commercial bus protocols, such as the PCI bus, to support the wider use of COTS hardware.

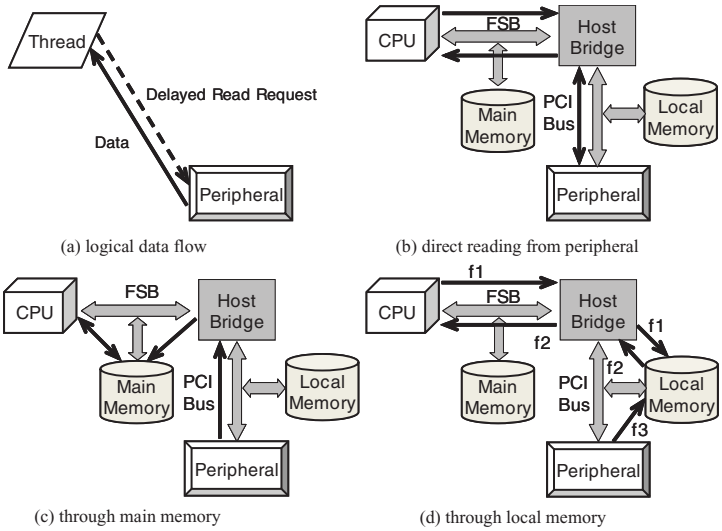
The general method of defining modeling blocks in AADL is to combine the standard components that AADL provides. However, AADL does not include every possible kind of component that we may encounter during development. For example, there are no bridges, switches, registers, or batteries. These nonstandard components can be represented by extending suitable component types in AADL that have a somewhat similar syntax. For example, bridges could be represented as `device` components, because bridges are connected to buses and should be considered as a hardware component. If no built-in types are suitable, there is always the `system` component, which can be used with almost any syntax. Note that bridges are shown as shaded squares, which represent `data` in AADL, in all the figures to easily differentiate from peripherals.

PCI bus is not designed for real-time systems. Since our approach is to develop modeling blocks corresponding to architecture building blocks for hard real-time applications, the selection of building blocks, configuration rules, and transaction types is important. After receiving hardware architecture models from industry companies, it became natural to define modeling blocks that represent the *PCI bridge* and *PCI bus*,



(a) example of bus architecture (b) example of alternative bus architecture

Fig. 3. Bus architecture examples.



(a) logical data flow (b) direct reading from peripheral

(c) through main memory (d) through local memory

Fig. 4. Input/output configuration example.

as shown in Figure 3(b). These are predefined in AADL so that users can use them or extend them. Using these modeling blocks lets us express any tree shape of PCI bus architecture and easily understand the architecture. AADL allows components to be *extended*. We use extension for checking the applicability of an analysis. An analysis could distinguish between a general and an extended component. When a model includes an extended component, the tool will first pass the extended component to the analysis. If the analysis does not recognize the extended component as an analysis-specific component, then the tool passes it as a general component to the analysis. In our examples, host bridge and Front-Side Bus (FSB) will be modeled with general bridge and bus components because ASIIST does not recognize any extension of these kinds (host bridge and FSB).

4.1.3. Logical Data-Flows in AADL. Logical data-flows are the exchange of data which is required by applications executing in a system. AADL plays an important role of specifying logical data-flows. Figure 4(a) is a graphical example of a logical flow specified using AADL. It is independent of the hardware through which the data is physically transmitted and any protocols that are applied to the hardware components. Properties

of a logical data-flow include but are not limited to data size, period, deadline, source, and destination.

The standard AADL already provides a reasonable syntax for expressing the interactions between software components by using `event`, `data`, or `event data ports` and `connections`. AADL `flows` can be specified to express a sequence of connections to trace data or control. Connections can also be bound to `buses` to analyze the effect of the I/O bus architecture.

4.1.4. Extension for Physical Flow Descriptions. A logical data-flow corresponds to a physical flow which passes through multiple hardware components. If the source and destination applications run on different CPUs that are located is separate systems, the flow must also navigate through a network. Depending on the type of the hardware used, the data is regulated to experience the already defined protocol for the COTS components it goes through. A designer can also make I/O configurations, as shown in Figure 4, to improve system performance. The real system performance is dependent on these physical flows that are derived from the logical data-flows and I/O configurations.

The standard AADL provides the `Actual_Connection_Binding` property to specify the physical path of a port connection. This property is instanced with a list of execution platform references, that specify the sequence of execution platform resources that are to be traversed by the logical data-flow. For the example shown in Figure 4(b), we could assign a list value of "Peripheral, PCI Bus, Bridge, FSB, CPU" to represent the physical flow path of the data.

However, just using the `Actual_Connection_ Binding` property is not expressive enough to represent the various multiple physical flows that will be used for a single data port connection. For Figure 4(c), we may try considering a list value of "Peripheral, PCI Bus, Bridge, FSB, System Memory, FSB, CPU" which somewhat expresses the fact that the data goes through the system memory. However, this is invalid for the standard AADL property `Actual_Connection_Binding` because it includes a `memory` reference which is not allowed. The description also does not specify in what period the peripheral writes to the system memory or what kind of PCI transaction it uses. We will now go on to extend AADL to address these issues in the description of physical flows within the PCI protocol.

4.1.5. Extensions to AADL for PCI Protocol. We have defined a set of properties that extend AADL to make it capable of describing the physical flows needed for our new analysis. Adding new properties to AADL is a common practice for tool developers because additional information needed may be missing in the standard.

Figure 5 describes the new properties that we have defined. We used the fact that the physical flows can be categorized into write transactions and read transactions. `Write_ Actual_Connection_Binding` and `Read_Actual_Connection_Binding` are used the same way as `Actual_Connection_Binding` except for the fact that they hold the list of execution platform components for each write or read physical flow. These connection binding property values can be derived from other information such as the logical connection and the I/O configuration settings and ASIIST supports the autogeneration of such list values. To support special hardware which may generate uncommon physical flows, we have defined these properties so that users can manually specify all cases and ASIIST can analyze them. Depending on the protocol that the PCI bus provides, `PCI_Write_Type` and `PCI_Read_Type` can have the values `Posted_Write`, or `Delayed_Write`, and `Delayed_Read` respectively. These are the transactions that can be supported for hard real-time analysis. Previous versions of PCI transactions, before 3.0, which utilize blocking should not be used for real-time systems because transaction delay can be extremely large in the worst case. `Delayed_Write` is used when an initiator would like to get acknowledgements that the write transaction has finished. An initiator is an entity

```
property set physicalflow is
 Write_Actual_Connection_Binding: inherit list of reference
    (bus, processor, device, memory, system) applies to
    (port connections, thread, thread group, process, system);
 PCI_Write_Type: inherit enumeration (Posted_Write, Delayed_Write)
    applies to (port connections, thread, thread group, process, system);
 PCI_Write_Period: inherit Time applies to
    (port connections, thread, thread group, process, system);

 Read_Actual_Connection_Binding: inherit list of reference (bus, . . . ;
 PCI_Read_Type: inherit enumeration (Delayed_Read) applies to . . . ;
 PCI_Read_Period: inherit Time applies to (port connections, . . . ;

 Network_Actual_Connection_Binding: inherit list of reference (bus, . . . ;
end physicalflow;
```

Fig. 5. Physical flow property definition.

that can start a bus transaction (either a peripheral or the CPU). Although AADL has a standard property named `Period` which is used to represent the period of a task, when there are multiple physical flows, each physical flow could have a different period due to I/O service implementations. For example, a DMA may be updating received data for a specific memory address more frequently than the task that is sending the data. Thus, we have also included the properties `PCI_Write_Period` and `PCI_Read_Period` to specify the period of each physical flow.

4.1.6. Extension for Network Flow Descriptions. When a logical data-flow goes through a network we would like to specify the actual datapath for the network as well. For a complex network, multiple paths may exist. The property named `Network_Actual_Connection_Binding` in Figure 5 will hold the list of network components to represent the path. In safety-critical hard real-time systems, the network path needs to be deterministic to be certifiable. Thus, this property is configurable and should be specified for a complete analysis.

4.2. Virtual Integration Using ASIIST for IMA

A tool that will read large system designs and build and solve complicated schedulability equations is critical to the development of hard real-time systems. ASIIST is a virtual integration tool that reads input from a system model specified in AADL and then analyzes the delays in the data-flows that take place on a customized hardware platform. The underlying algorithm is modular and capable of analyzing alternative system architectures represented by the composition of the modeling blocks and its data-flows. In Section 4.1, we defined hardware modeling blocks that have a natural one-to-one correspondence with actual hardware components. Bus communications have also been specified in detail to capture the true nature of the bus flows that take place on a hardware platform. These details are sufficient to allow ASIIST to run its bus delay analysis.

The analysis algorithm that ASIIST uses to determine the internal bus delay is based on network calculus theory [Boudec and Thiran 2001], which can be used to determine delay bounds for network traffic, and also to model different types of real-time systems [Thiele et al. 2000]. Depending on whether the type of arbitration used in implementing the bridges is (FIFO or round-robin) the pessimism of the analysis can be adjusted. Protocol-specific properties are also recognized for PCI. More details about the analysis are available in a technical report [Pellizzoni et al. 2008].

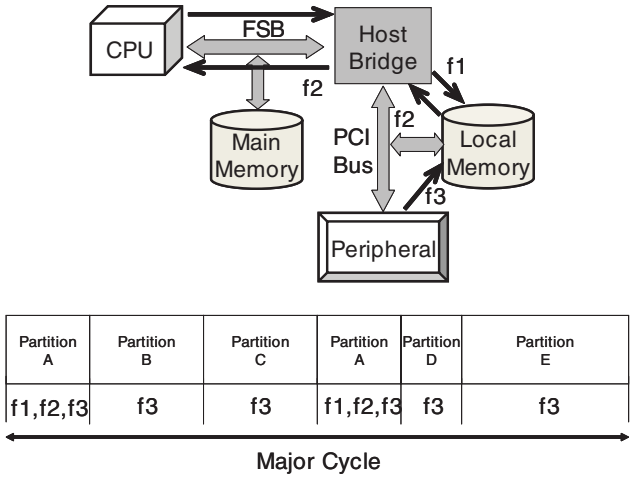


Fig. 6. Physical flow across partitions.

As mentioned before, a designer could specify a different I/O configuration (Figure 4) to improve system performance. System performance is dependent on the physical flows that are derived from logical data-flows and I/O configurations. Physical flows are defined by the sequence of hardware components through which the data actually flows. In Figure 4(d), there are three physical flows, corresponding to: the request message that is initiated by an application executed on the CPU (f1); the data that is sent from the local memory to the processor or main memory (f2); and the data sent from a peripheral to local memory (f3).

In an IMA environment, ASIIST characterizes logical data-flows and infers their effects on each partition based on the type of component that is initiating a bus transaction. If the component is an application running on a CPU, any physical flow that the application directly initiates (f1) or requests (f2) is only considered to exist in the same partition as the application. However, if a general peripheral initiates a bus transaction (f3), the physical flow is considered to exist across all partitions in the system, unless additional restrictions are specifically applied. Thus, if the application which requires these data-flows is assigned to partition A, the physical flows are those described in Figure 6. If an application is assigned to one partition, but requires data from elsewhere, like f3, the eventual effect will be to increase the data traffic across all partitions for the bus components through which f3 passes.

This "knock-on" effect from one partition to another is understood by system developers, but it is very difficult to account for it using conventional testing methods, because each partition of an IMA system is developed by separate entities and tested in isolation; other physical flows produced by other applications in different partitions appear not to exist and become untestable. Thus, using a tool that supports IMA, such as ASIIST, is valuable during the phase of virtual integration.

4.3. End-System Model and Demonstration

In this section we show how ASIIST can be used for the worst-case delay analysis of end-systems. We consider the sending end-system to be a Line-Replaceable Unit (LRU), which is a potentially complicated unit or component used in an airplane, spacecraft, or ship. The CPUs in the LRUs generally have only one task to perform. LRUs reduce the system development costs and also maintenance times, because they are sealed units which can be replaced quickly.

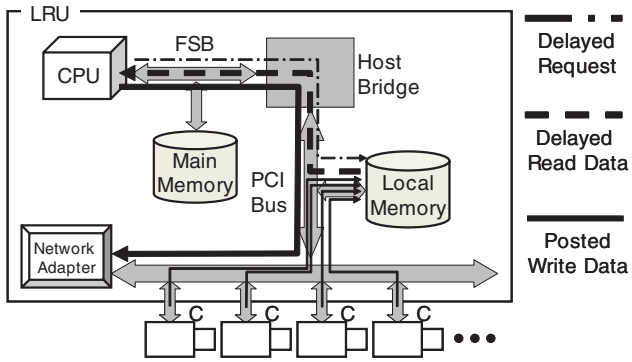


Fig. 7. LRU structure and connected cameras in the sending end-system.

Discription	Source	Destination	Period(sec)	Delay(sec)	Data Size(Byte)	PCI Transaction
Connection	lruThread	Network_Adapter	0.01	0.0001183508201300828	2500	
PCI PostedWrite	vm0	Network_Adapter	0.01	0.00003870728499686127	2500	postedWrite
PCI DelayedRead	localMemory	vm0	0.01	0.00007964353513322153	2500	delayedRead
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625	
PCI PostedWrite	camera1	localMemory	0.01	0.00003771805909162345	625	postedWrite
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625	
PCI PostedWrite	camera2	localMemory	0.01	0.00003771805909162345	625	postedWrite
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625	
PCI PostedWrite	camera3	localMemory	0.01	0.00003771805909162345	625	postedWrite
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625	
PCI PostedWrite	camera4	localMemory	0.01	0.00003771805909162345	625	postedWrite

Fig. 8. LRU bus delay for 4 cameras.

Discription	Source	Destination	Period(sec)	Delay(sec)	Data Size(Byte)	PCI Transaction
Connection	lruThread	Network_Adapter	0.01	0.0002374226519330825 4	5000	
PCI PostedWrite	vm0	Network_Adapter	0.01	0.00007755586792452831	5000	postedWrite
PCI DelayedRead	localMemory	vm0	0.01	0.00015986678400855425	5000	delayedRead
Connection	sensorThread_send	sensorThread_rcv	0.01	7.57217258242625E-5	625	
Connection	sensorThread_send	sensorThread_rcv	0.01	7.57217258242625E-5	625	

Fig. 9. LRU bus delay for 8 cameras.

Figure 7 is an example of an LRU model that we will use. This model is composed of a CPU, a main memory, a local memory, and a network adapter. In our monitoring application, cameras are connected to the LRU unit, and the visual data are sent from the cameras and saved in the local memory. Local memory is used to temporarily save the data because it is too large to be saved in the main memory. The number of cameras that are connected to one LRU is a design problem because if we have less cameras, the number of LRUs will increase for the same amount of total cameras which will lead to increased cost in maintenance and weight. If there are more cameras, the local delay of the visual data will increase and may cause a problem. The arrows in Figure 7 represent the physical flows depending on the protocol they use, which is PCI in our case. An application reads in data from the local memory for aggregation and then sends it through the network adapter.

Figures 8 and 9 show screen shots of the output from ASIIST when the LRU has 4 or 8 cameras connected. ASIIST provides table and graphical views (Figure 10) of the output to assist the user in evaluating problematic connections and finding congested

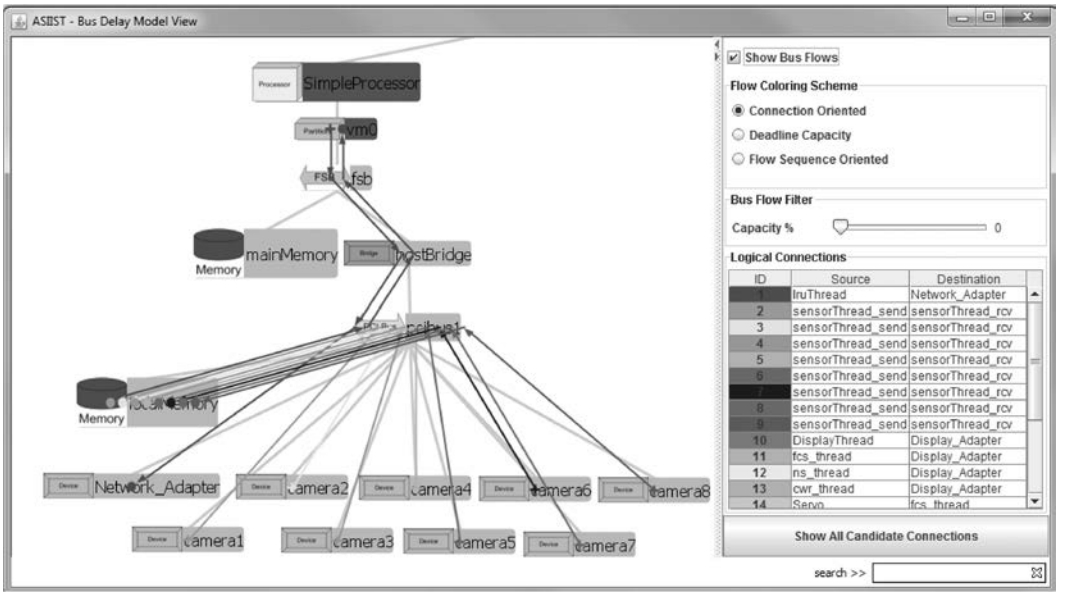


Fig. 10. LRU bus delay graphical view.

Table I. Application Data Receive Parameters

Application	Sensor	Network Adapter	Memory	Data Size (Byte)	Period
EMS	Camera	Network Adapter 2	Local	12500	10 ms
FCS	Servo	Network Adapter 2	Main	32	20 ms
	AHRS	Network Adapter 2	Main	32	20 ms
	Navigation Radar	Network Adapter 2	Main	32	20 ms
	Flight Control Panel	Network Adapter 2	Main	32	20 ms
CWR	Color Weather Radar	Network Adapter 1	Local	1024	100 ms
NS	Gyroscope	Network Adapter 2	Main	32	20 ms
	Accelerometer	Network Adapter 2	Main	32	20 ms
	GPS Receiver	Network Adapter 2	Main	32	20 ms
	Doppler Radar	Network Adapter 2	Main	32	20 ms

AHRS: Attitude and Heading Reference Systems

bus components. Users can interact with the graphical view to see the subset of flows that they are interested in by selecting multiple filters.

As for the receiving end-system, we use the bus architecture examples introduced in Figure 3 which are composed of a CPU, a main memory, local memory, two network adapters, a display adapter, and PCI components. The CPU has four applications (Environmental Monitoring System (EMS), FCS, CWR system, and the Navigation System (NS)) running on different partitions. Each application receives data from multiple sensors, as shown in Table I. Note that because these data arrive through the network at an undetermined time, we have to assume that each data can arrive during any partition. The sensor data arrives through their assigned network adapters (1 or 2). Applications such as the EMS and CWR save the data in a local memory which is much larger but slower than the main memory. After processing the received data, all applications need to send display data through the display adapter to be seen by the pilot, as described in Table II. Using these parameters, we will use ASIIST to analyze the bus delay of the receiving end-system in Section 7.

Table II. Application Data Send Parameters

Application	Partition	Destination	Data Size (Byte)	Period
EMS	vm0	Display Adapter	125000	100 ms
FCS	vm1	Display Adapter	1024	100 ms
CWR	vm2	Display Adapter	1024	100 ms
NS	vm3	Display Adapter	1M	100 ms

5. NETWORK DELAY ANALYSIS

While most previous work in the research community has been developing high-speed switches and routers and also finding worst-case delay bounds for real-time traffic, these methods does not guarantee isolation. When one traffic changes, the whole network of systems needs to be checked as to whether the newly established bounds are valid.

Currently, the ARINC 664 requires isolation by having traffic prioritization which guarantees that lower-priority frames would not preempt a higher-priority frame. This will guarantee that a lower-priority application will not cause latency changes to a higher-priority application. While this may be sufficient for safety guarantee, this requirement disregards what happens in the real industry. When incremental traffic is added to the switch, we do not know how much other network connections would start to miss its max delay and be retransmitted or handled at the end-systems. ARINC 664 requires a max delay parameter defined for each network connection. When a frame is not transmitted within the max delay, it must be discarded in the switch. Thus, when a new networking feature is added, any system that has a lower-priority frame but only shares the network would still need to be reevaluated for its change in network latency and dropped frame rate. In many cases, this is already a lot of reevaluation in a large network. It would be valuable to approach this incremental development issue in a way where delay can be guaranteed for the switch and not just be checked and dropped.

5.1. Real-Time Switching Algorithm

Packet switching in intermediate switch systems helps achieve high-speed data transmission, data rate/format transparency, and configurability, which are necessary for making networks future-proof [Yao et al. 2000]. The following three points must be considered when designing real-time switches which are favorable towards incremental development. First, the worst-case switching delay must be predictable to guarantee that the timing constraints for real-time applications are met. Second, when adding new network traffic, the worst-case switching delay of existing traffic must be guaranteed not to change until a certain provable condition is invalidated. We propose a real-time switching algorithm that supports these characteristics. Third, the packet size should be sufficiently large so that the control overhead is reduced to a reasonably low value. We determine the heuristic optimal packet size by simulation experiments using the proposed switching algorithm.

Considering these three aspects, we design a real-time switching algorithm that can guarantee a bounded delay with any feasible traffic. The underlying hardware fabric that we adopt is $N \times N$ Virtual-Output Queue (VOQ) crossbar [Peterson and Davie 2000]. The data bus from each input intersects with the data bus of each output. The intersections can be turned on or off during runtime by the switch scheduling logic. To facilitate the design of the switching logic, crossbar switches transfer packets in fixed-size fragments called time slots. Therefore, the scheduling logic works periodically: it determines a matching between inputs and outputs at the beginning of each time slot; then all scheduled packets are transferred synchronously across the crossbar fabric, taking one time slot; and then the next period starts, so on and so forth. We will

also use the widely adopted VOQ architecture, where each input maintains a virtual output queue for each output. VOQs eliminate head of line blocking [Wang et al. 2008], but packets from different inputs' VOQs still contend for the same output. Thus it is required to reduce this contention, so as to improve the hardware utilization.

5.1.1. Clock-Driven Scheduling as a Virtual Machine Task. A widely used approach to the scheduling of real-time virtual machine tasks (VM-tasks) [Liu 2000; Davis and Burns 2005; 2006; Deng and Liu 1997; Kuo and Li 1999; Lipari and Bini 2003; Shin and Lee 2003] is clock-driven scheduling [Liu 2000]. Suppose a VM task $(\mathcal{P}, C)$ is served C times during each clock period of $\mathcal{P}$. Let f_k^{ij} denote the kth real-time flow from input port I_i to output port O_j, where $k = 1, 2, \ldots, K_{ij}$, and K_{ij} is the total number of flows from I_i to O_j. The flow f_k^{ij} is additionally associated with a VM task $(\mathcal{P}, C_{ijk})$, because f_k^{ij} has C_{ijk} packets to be served. Using clock-driven scheduling, the delay to f_k^{ij} will be bounded as long as the switching algorithm can guarantee a worst-case bound on the time required to forward all the C_{ijk} packets of the flow f_k^{ij}.

Let C_{ij} denote the total number of packets to be forwarded from I_i to O_j, so that $C_{ij} = \sum_{k=1}^{K_{ij}} C_{ijk}$. For the set of VM tasks $\{(\mathcal{P}, C_{ijk})\}$, $i = 1, \ldots, N$, $j = 1, \ldots, N$, $k = 1, \ldots, K_{ij}$, we introduce a feasibility condition for the real-time traffic during each clock period of $\mathcal{P}$, which contains L time slots, as follows.

$$\sum_{j=1}^{N} C_{ij} \le L, i = 1, 2, \ldots, N \tag{2}$$

$$\sum_{i=1}^{N} C_{ij} \le L, j = 1, 2, \ldots, N \tag{3}$$

The *feasible traffic* is defined as the traffic that satisfies a feasibility condition, and it should be noted that infeasible traffic which does not meet these conditions is naturally unschedulable, which refers to the situation in which some VM tasks will miss their deadlines.

5.1.2. Clearance-Time-Optimal Switching Policies. In order to design a switching algorithm with guaranteed delay, we introduce clearance-time-optimal policies. In this setting, assuming $Q_{ij}(t)$ denote the current number of queued packets in queue (i, j) at time slot t, if every queue in the switch has initial packets of $Q_{ij}(0)$, and has no further arrivals, we call it a *one-shot traffic*. The clearance time is then defined as the the number of time slots required to serve every packet in the switch. A switching policy that minimizes the clearance time is called a clearance-time-optimal policy.

Because no more than one packet can be switched at any port of a switch in one time slot due to the crossbar constraint, an obvious lower bound on the clearance time T_{clear} is the maximum number of packets waiting at any input port.

$$T_{\text{clear}} \ge \max\left(\max_i \sum_{j=1}^{N} Q_{ij}(0), \max_j \sum_{i=1}^{N} Q_{ij}(0)\right) \tag{4}$$

It is apparent that this bound is tight, and that the minimum clearance time T_{clear}^* is equal to the right-hand side of Eq. (4).

How do we design a switching algorithm that can achieve this minimum clearance time T_{clear}^*? We first introduce a *critical-port policy* as follows: given a bipartite graph G, port i is *critical* if its weight, which is the length of its queue, is no smaller than that of any other port; and a critical-port matching M serves every critical port.

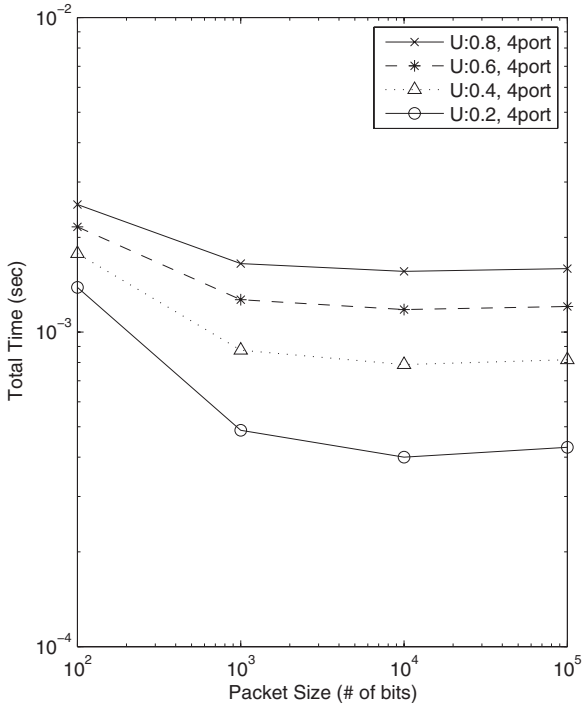


Fig. 11. Switching delay of one-shot traffic including switching overhead.

Consequently, a critical-port scheduling policy must generate a critical-port matching at every time slot. It has been shown that a critical-port policy is also clearance-time optimal [Gupta et al. 2009].

5.1.3. Design of a Real-Time Switch with Clock-Driven Scheduling. We can now use the concept of clearance time and the critical-port policies described in the previous section to design a switching algorithm that can guarantee a bounded delay for any feasible traffic. Our particular goal is to produce a switch design framework that can achieve this bound by combining clearance-time-optimal scheduling with the VM task architecture and the critical-port policy introduced in the previous section, in the following manner: the traffic arriving during each clock period is buffered and served in the next clock period. This achieves one-shot traffic, which is the basic requirement for clearance-time-optimal scheduling: clearance-time optimality then guarantees that any feasible traffic is served in two clock periods. In effect, we accept an additional delay of one clock period, to ensure a deterministic delay bound of $2\mathcal{P}$.

Consequently, our switch architecture can support real-time traffic because the deterministically bounded delay is more critical for real-time applications than the average delay performance. In addition, this also guarantees that preexisting network traffic will maintain their delay bound until the added traffic makes the set of traffic going through the switch infeasible, which favors incremental development. By adjusting the size of the clock period, we can modify the size of the guaranteed delay bound. However, if the clock period is too short, then assuming the same packet size, the value of L decreases and the network likely becomes infeasible when new network traffic is added. We can provably check if the guaranteed delay bound is still valid or not when using our new real-time switching algorithm.

Table III. Simulation Parameters

Parameters	Value
Packet Size	10 Kb
Port Capacity	1 Gb/s
Clock Period ($\mathcal{P}$)	1 ms
The Number of Time-Slots in One Clock Period (L)	100
Switching Overhead (δ_o)	100 ns

5.2. Packet Size Evaluation

Among many possible realizations of critical-port policies, we adopt Lazy Heaviest-Port First (LHPF) matching, which we will now explain: first, the threshold th of a matching M is defined as the lowest integral weight for which M matches all the ports. For example, a perfect matching that switches every port with packets in its queue has $th = 1$. An LHPF matching has the lowest threshold of all possible matchings.

In order to investigate the effect of packet size on the performance of switch implemented in conjunction with this LHPF matching, the switching delay is measured for various packet sizes. The simulation results in Figure 11 show the expected switching delay as a function of the packet size for different values of demand utilization[1] U, using the simulation parameters presented in Table III. Note that the switching overhead, which is the time to be taken for reconfiguration of the switch fabric, is assumed to be 100 ns [Papadimitriou et al. 2003], the number of input/output ports and utilization are 4 and 0.2, respectively, and each point is the average of the values obtained in 10,000 simulation runs. Clearly, within a fixed clock period, the total number of time slots would decrease when the packet size increases at a given capacity per port. Hence, if the packet size increases, the switching overhead is expected to decrease. However, our simulation results in Figure 11 contradict the predicted results. It first decreases with an increase in the packet size, but shows a slight increase when the packet size becomes 10^5 bits. This is because an increase in the packet size may result in decreased granularity, which is indicative of a decrease in the number of time slots in one clock period. Thus, to minimize the effect of switching overhead, packet size needs to be selected adequately. On the basis of simulation results, we determine the heuristic optimal packet size to be 10^4 bits.

6. END-TO-END LATENCY ANALYSIS

By having a networked system that uses our switching algorithm for its switches we can determine the worst-case bound of the end-to-end latency ($\bar{\delta}$) as

$$\begin{aligned}\bar{\delta} &= \delta_I + \delta_S + \delta_O \\ &= \delta_I + \sum_{k=1}^{h} \delta_{S_k} + \delta_O \le \delta_I + h \cdot 2(\mathcal{P} + L\delta_o) + \delta_O, \end{aligned} \tag{5}$$

where δ_S is the total worst-case delay δ_S between the network adapter on the sender side and another adapter on the receiver side, using a route that includes several switches from the data network. Since the propagation delays between systems are negligible, it can be simplified to $\delta_S = h \cdot 2(\mathcal{P} + L\delta_o)$, where h denotes the hop count, δ_o is the switching overhead, and $\mathcal{P}$ is the clock period that contains L time slots, because the proposed switch guarantees any feasible real-time traffic to be switched in two clock periods.

[1]The ratio between the average per-port traffic load and the per-port capacity.

Table IV. Simulation Parameters of the Network System

Parameters	Value
FSB Bandwidth	19.2 Gb/s
PCI Bus Bandwidth	1.064 Gb/s
The Max Size of Data Allowed by Bridge	1 Kb
Packet Size	10 Kb
Port Capacity	1 Gb/s
Clock Period ($\mathcal{P}$)	1 ms
The Number of Time-Slots in One Clock Period (L)	100
Switching Overhead (δ_o)	100 ns
Captured Imaging Data per Camera	0.5 Mb/s
Data Fetching Period of CPU for Display	10 ms

Each logical data connection in an AADL model will have the set of AADL properties that represents the hardware or the network flow, as explained in Section 4.1. ASIIST will analyze each delay (δ_I, δ_S, δ_O) separately and show the total worst-case end-to-end latency.

7. CASE STUDY EVALUATION

As described in Figure 1, we examine the case where we would like to add five LRUs to an already existing network system. The data network is shared with other sensory data traffic. In the front side of the aircraft, we have the receiving end-system which is a computational node that processes the data received from the cameras of the LRUs. We assume that the receiving end-system also has other applications (FCS, CWR system, and NS) executing on its CPU. Other computational nodes may be connected to the network to share the network bandwidth. However, as long as we have a feasible traffic in every switch, we do not have to be concerned about other nodes because their worst-case end-to-end latency will not be changed while using our switching algorithm. This is one of the major benefit of our switching algorithm because it simplifies the design process of adding new networking features while maintaining the robustness of unrelated components in a network system. We only need to analyze the end-systems where we add the new features and check the feasibility of the network. We will assume that the end-systems are schedulable, which can be analyzed by ASIIST as well [Mohan et al. 2009].

By using an AADL model which models the logical data-flows and parameters presented in Table I and II for the receiving end-system, we run ASIIST to check if there are any problematic connections. Table IV shows other parameters of the data network and the common hardware components inside the end-systems.

As can be seen in Figure 12, the delay for the connection between the Network_Adapter2 to the EMSThread is about 0.0024 sec. We can expect to decrease this delay by changing the simple bus architecture described in Figure 3(a) to the alternative in Figure 3(b). Peripherals are no longer connected to one common PCI bus. Since the camera data is relatively large (12500 Bytes) and arrives at a high frequency, they need to be saved in the local memory. By changing the bus architecture of the receiving end-system to Figure 3(b), we can reduce the delay for the connection between Network_Adapter2 and EMSThread to 0.00166 sec (Figure 13). This is due to the fact that now the data-flow to the local memory and the data-flow to the display adapter do not interfere with each other. The cost we pay is that other flows have longer delays due to the lengthened bus architecture.

The results in Table V summarize the predicted worst-case delays within the receiving end-system, across the data network depending on the number of hops, and the LRUs. The network delay is based on simulation using the parameters

Discription	Source	Destination	Period(sec)	Delay(sec)	Data Size(Byte)
Connection	Network_Adapter2	EMSThread	0.01	0.0024240964428309322	12500
Connection	IruThread	Network_Adapter	0.01	0.0001183508201300828	2500
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	EMSThread	Display_Adapter	0.1	0.0011057407029920038	125000
Connection	fcs_thread	Display_Adapter	0.1	1.1291792207030274E-4	1024
Connection	ns_thread	Display_Adapter	0.1	1.1291792207030274E-4	1024
Connection	cwr_thread	Display_Adapter	0.1	0.00811290283665475 4	1000000
Connection	Servo	fcs_thread	0.02	0.0011163900667086305	32
Connection	AHRS	fcs_thread	0.02	0.0011163900667086305	32
Connection	NavigationRadar	fcs_thread	0.02	0.0011163900667086305	32
Connection	Gyroscope	ns_thread	0.02	0.0011163900667086305	32
Connection	Accelerometer	ns_thread	0.02	0.0011163900667086305	32
Connection	GPSrcv	ns_thread	0.02	0.0011163900667086305	32
Connection	DopplerRadar	ns_thread	0.02	0.0011163900667086305	32
Connection	ColorWeatherRadar	cwr_thread	0.1	0.0102465690994677	1024
Connection	FCP	fcs_thread	0.1	0.0011163900667086305	32

Fig. 12. Receiving end-system bus delay with bus architecture (a) (4 cameras).

Discription	Source	Destination	Period(sec)	Delay(sec)	Data Size(Byte)
Connection	Network_Adapter2	EMSThread	0.01	0.001663147367052421	12500
Connection	IruThread	Network_Adapter	0.01	0.0001183508201300828	2500
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	sensorThread_send	sensorThread_rcv	0.01	3.771805909162345E-5	625
Connection	EMSThread	Display_Adapter	0.1	0.0019496246996906373	125000
Connection	fcs_thread	Display_Adapter	0.1	3.327119698229773E-5	1024
Connection	ns_thread	Display_Adapter	0.1	3.327119698229773E-5	1024
Connection	cwr_thread	Display_Adapter	0.1	0.0154749758267095	1000000
Connection	Servo	fcs_thread	0.02	0.0011144551097106393	32
Connection	AHRS	fcs_thread	0.02	0.0011144551097106393	32
Connection	NavigationRadar	fcs_thread	0.02	0.0011144551097106393	32
Connection	Gyroscope	ns_thread	0.02	0.0011144551097106393	32
Connection	Accelerometer	ns_thread	0.02	0.0011144551097106393	32
Connection	GPSrcv	ns_thread	0.02	0.0011144551097106393	32
Connection	DopplerRadar	ns_thread	0.02	0.0011144551097106393	32
Connection	ColorWeatherRadar	cwr_thread	0.1	0.011138622382646058	1024
Connection	FCP	fcs_thread	0.1	0.0011144544965064303	32

Fig. 13. Receiving end-system bus delay with bus architecture (b) (4 cameras).

summarized in Table IV. Recall that the heuristic optimal packet size is 10^4 bits according to the results presented in Figure 11. The data flowing from LRU 3 to the receiving end-system pass through three switches. Thus, the delay in the data network is three times that in the case of LRU 1 or LRU 5, in which data pass only through one switch. The resulting worst-case end-to-end latency expected from LRU 1 (or LRU 5) is 3.97081 ms, and that from LRU 2 (or LRU 4) is 4.1728 ms; the worst-case end-to-end delay from LRU 3 is 4.3748 ms.

We can easily check the end-to-end latency for 8 cameras per LRU by changing the model. This would enhance the environmental monitoring feature by having more cameras and thus being able to conceive a larger area far away when zooming. Table VI shows the result for 8 cameras. One will notice that the delay in the receiving end-system increases because a larger data is received from the additional cameras.

Table V. The Worst-Case End-to-End Latency Analysis Results (4 cameras)

Delay Components		$h=1$	$h=2$	$h=3$
δ_I	Camera to Local Memory	0.038		
	Local Memory to CPU	0.079		
	CPU to Network Adapter	0.039		
$h \cdot 2(\mathcal{P} + L\delta_o)$	Across Switches	0.2020	0.4040	0.6060
δ_O	Network Adapter to Local Memory	0.1038		
	Local Memory to CPU	1.559		
	CPU to Display Adapter	1.950		
$\bar{\delta}$	The Worst-Case End-to-End Latency	3.9708	4.1728	4.3748

(Unit: ms)

Table VI. The Worst-Case End-to-End Latency Analysis Results (8 cameras)

Delay Components		$h=1$	$h=2$	$h=3$
δ_I	Camera to Local Memory	0.0757		
	Local Memory to CPU	0.1598		
	CPU to Network Adapter	0.0775		
$h \cdot 2(\mathcal{P} + L\delta_o)$	Across Switches	0.2020	0.4040	0.6060
δ_O	Network Adapter to Local Memory	0.1978		
	Local Memory to CPU	2.053		
	CPU to Display Adapter	1.950		
$\bar{\delta}$	The Worst-Case End-to-End Latency	4.7158	4.9178	5.1198

(Unit: ms)

However, the delay across the data network does not change unless the additional data makes one of the switches infeasible or the topology of the network has changed. By using a virtual integration tool, such as ASIIST, we can quickly see how adding more cameras affects the end-to-end latency across the network and inside the end-systems.

8. CONCLUSION

This article presents an automated tool, ASIIST, and a new real-time switching algorithm to model and analyze end-to-end latency for network systems. We used a case study of incrementally adding a new networking feature to an already operational network which is what really happens in the industry. Early analysis can only be achieved by virtual integration tools which captures the true characteristics of IMA systems. We demonstrate that by using ASIIST one can safely decide high-level design problems (system architecture) and prevent them from causing problems later in the design process. Real-time properties such as the end-to-end latency are really difficult to calculate by hand and eventually prevent the designer from trying different bus architectures and network topologies. Using our new real-time switch greatly assists in the incremental development process by guaranteeing and knowing whether the worst-case end-to-end latency of other system modules that share the network have changed or not. The combined usage of ASIIST and our real-time switch provides early modularized predictive analysis which simplifies the analysis of the overall network system for ease of understanding. By no means will this replace rigorous testing after real integration, but it will enhance the designer's capability to make more promising decisions. Currently we have used AADL version 1.0. With the recent new version release of AADLv2, some changes will be made to accommodate the new features available in AADLv2. First of all, the `Actual_Connection_Binding` can now include memory

components. The newly introduced component type called `virtual bus` can be included in models to represent communication protocols or virtual channels. In this manner, virtual buses can represent different types of protocol-specific bus transactions so that they can be bound to AADL connections. Commonly used and allowed virtual bus types will be defined for each hardware platform so that any connection that goes through the hardware platform would have to select from one of the available virtual buses. Virtual bus types can inherently represent read and write transactions which makes some properties that we defined (`PCI_Write_Type` and `PCI_Read_Type`) redundant. AADL property definitions have become more expressive with AADLv2, so that properties can be defined as a list of another AADL property. This lets us represent an array of connection binding lists. For complex I/O configurations, such as using special I/O control devices, where there are multiple read and write transactions that exist for a single logical data-flow connection, an array of read and write connection binding properties will cover all the physical flows. Also, a set of properties can be grouped as a `record` type so that users would know certain properties are related. Implementation of these features to easily manipulate the I/O specification is left for future work.

REFERENCES

PCI-SIG. 2012. http://www.pcisig.com/specifications.

Aeronautical Radio Inc. 1991. Design guidance for integrated modular avionics. ARINC rep. 651.

Aeronautical Radio Inc. 2005. Aircraft data network, Part 7 - Avionics full duplex switched ethernet network. ARINC rep. 664P7-1.

Audsley, N. and Wellings, A. 1996. Analysing apex applications. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*. IEEE, Los Alamitos, CA, 39–39.

Baker, T. G. 2002. Lessons learned integrating COTS into systems. In *Proceedings of the Ist International Conference on COTS-Based Software Systems (ICCBSS'02)*. Springer, Berlin, 21–30.

Binns, P., Englehart, M., Jackson, M., and Vestal, S. 1996. Domain specific software architectures for guidance, navigation and control. *Int. J. Softw. Engin. Knowl. Engin. 6,* 2, 201–227.

Boudec, J.-Y. L. and Thiran, P. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer.

Davis, R. and Burns, A. 2005. Hierarchical fixed priority preemptive scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS'05)*. IEEE, Los Alamitos, CA, 389–398.

Davis, R. and Burns, A. 2006. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*. IEEE, Los Alamitos, CA, 257–270.

Deng, Z. and Liu, J. W.-S. 1997. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*. IEEE, Los Alamitos, CA, 308–319.

Driscoll, K. and Hoyme, K. 1992. The airplane information management system: An integrated real-time flight-deck control system. In *Proceedings of the 13th IEEE Real-Time Systems Symposium (RTSS'92)*. IEEE, Los Alamitos, CA, 267–270.

Gopalakrishnan, S., Sha, L., and Caccamo, M. 2004. Hard real-time communication in bus-based networks. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS'04)*. IEEE, Los Alamitos, CA, 405–414.

Gupta, G. R., Sanghavi, S., and Shroff, N. B. 2009. Node weighted scheduling. In *Proceedings of the 11th Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics'09)*. ACM, New York, 97–108.

Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., and Ernst, R. 2005. System level performance analysis - the SymTA/S approach. *IEE Proc. Comput. Digit. Techn. 152,* 2, 148–166.

Hoyme, K. and Driscoll, K. 1992. SAFEbus. In *Proceedings of the 11th IEEE/AIAA Digital Avionics Systems Conference (DASC'92)*. IEEE, Los Alamitos, CA, 68–73.

Hugues, J., Zalila, B., Pautet, L., and Kordon, F. 2008. From the prototype to the final embedded system using the ocarina AADL tool suite. *ACM Trans. Embedd. Comput. Syst. 7,* 4, 1–25.

Kuo, T.-W. and Li, C.-H. 1999. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*. IEEE, Los Alamitos, CA, 256–267.

LEE, Y.-H., KIM, D., YOUNIS, M., AND ZHOU, J. 2000. Scheduling tool and algorithm for integrated modular avionics systems. In *Proceedings of the 19th IEEE/AIAA Digital Avionics Systems Conference (DASC'00)*. Vol. 1. IEEE, Los Alamitos, CA, 1C2/1–1C2/8.

LIPARI, G. AND BINI, E. 2003. Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*. IEEE, Los Alamitos, CA, 151–158.

LIU, J. W.-S. 2000. *Real-Time Systems*. Prentice Hall.

MCKEOWN, N. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Trans. Netw. 7,* 2, 188–201.

MOHAN, S., NAM, M.-Y., PELLIZONI, R., SHA, L., BRADFORD, R., AND FLIGINGER, S. 2009. Rapid early-phase virtual integration. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*. IEEE, Los Alamitos, CA, 33–44.

NEELY, M. J., MODIANO, E., AND CHENG, Y.-S. 2007. Logarithmic delay for $n \times n$ packet switches under the crossbar constraint. *IEEE/ACM Trans. Netw. 15,* 3, 657–668.

PAPADIMITRIOU, G. I., PAPAZOGLOU, C., AND POMPORTSIS, A. S. 2003. Optical switching: switch fabrics, techniques, and architectures. *J. Lighwave Technol. 21,* 2.

PELLIZZONI, R., NAM, M.-Y., BRADFORD, R. M., AND SHA, L. 2008. A network calculus based analysis for the PCI bus. Tech. rep., University of Illinois at Urbana-Champaign, http://ece.uwaterloo.ca/ rpellizz/techreps/busanal.pdf.

PETERSON, L. L. AND DAVIE, B. S. 2000. *Computer Networks: A System Approach*. Morgan Kaufmann.

PORTER, J., KARSAI, G., VÖLGYESI, P., NINE, H., HUMKE, P., HEMINGWAY, G., THIBODEAUX, R., AND SZTIPANOVITS, J. 2009. Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation. Lecture Notes in Computer Science, vol. 5421, Springer, 20–34.

REXFORD, J., HALL, J., AND SHIN, K. G. 1998. A router architecture for real-time communication in multicomputer networks. *IEEE Transactions on Computers 47,* 10, 1088–1101.

REYNOLDS, B. 1998. An application of COTS Ethernet for avionics. In *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC'98)*. Vol. 2. IEEE, Los Alamitos, CA, J13/1–J13/8.

SAE. 2009. Architecture Analysis & Design Language (AADL). AS5506A. http://standards.sae.org/as5506a.

SCHARBARG, J.-L., RIDOUARD, F., AND FRABOUL, C. 2009. A probabilistic analysis of end-to-end delays on an AFDX avionic network. *IEEE Trans. Indust. Info. 5,* 1, 38–49.

SCHUSTER, T. AND VERMA, D. 2008. Networking concepts comparison for avionics architecture. In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference (DASC'08)*. IEEE, Los Alamitos, CA, 1.D.1–1–1.D.1–11.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Real-time scheduling support in futurebus+. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS'90)*. IEEE, Los Alamitos, CA, 331–340.

SHIN, I. AND LEE, I. 2003. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*. IEEE, Los Alamitos, CA, 2–13.

SINGHOFF, F., LEGRAND, J., NANA, L., AND MARCÉ, L. 2005. Scheduling and memory requirements analysis with AADL. *ACM SIGAda Ada Lett. XXV,* 4, 1–10.

SOKOLSKY, O., LEE, I., AND CLARKE, D. 2009. Process-algebraic interpretation of AADL models. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'09)*. Springer, 222–236.

THIELE, L., CHAKRABORTY, AND S., NAEDELE, M. 2000. Real-time calculus for scheduling hard real-time systems. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'00)*. Vol. 4. IEEE, Los Alamitos, CA, 101–104.

VENKATRAMANI, C. AND CHIUEH, T. 1997. Design and implementation of a real-time switch for segmented Ethernets. In *Proceedings of IEEE International Conference on Network Protocol (ICNP'97)*. IEEE, Los Alamitos, CA, 152–161.

WANG, Q., GOPALAKRISHNAN, S., LIU, X., AND SHA, L. 2008. A switch design for real-time industrial networks. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*. IEEE, Los Alamitos, CA, 367–376.

WELLER, T. AND HAJEK, B. 1997. Scheduling nonuniform traffic in a packet-switching system with small propagation delay. *IEEE/ACM Trans. Netw. 5,* 6, 813–823.

YAO, S., MUKHERJEE, B., AND DIXIT, S. 2000. Advances in photonic packet switching: an overview. *IEEE Comm. Mag. 38,* 2, 84–94.

Received February 2010; revised August 2010; accepted January 2011