Real-Time Control of I/O COTS Peripherals for Embedded Systems

Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
{sbak2, ebetti, rpelliz2, mcaccamo, lrs}@illinois.edu

Abstract

Real-time embedded systems are increasingly being built using commercial-off-the-shelf (COTS) components such as mass-produced peripherals and buses to reduce costs, time-to-market, and increase performance. Unfortunately, COTS interconnect systems do not usually guarantee timeliness, and might experience severe timing degradation in the presence of high-bandwidth I/O peripherals. To address this problem, we designed a real-time I/O management system comprised of 1) real-time bridges, and 2) a reservation controller. The proposed framework is used to transparently put the I/O subsystem of a COTS-based embedded system under the discipline of real-time scheduling. We also discuss computing a delay bound for I/O data transactions and determining worst-case buffer size. Finally, we demonstrate experimentally that our prototype real-time I/O management system successfully prioritizes I/O traffic and guarantees its timeliness.

1. Introduction

Integrating high-speed commercial-off-the-shelf (COTS) peripherals within a real-time system offers substantial benefits in terms of cost reduction, time-to-market, and overall performance. Due to mass production, COTS components are significantly cheaper to produce than their application-specific peers. Since COTS components are already designed, a system's time-to-market can be reduced by reusing existing components instead of creating new ones. Additionally, overall performance of mass produced components is often significantly higher than custom made systems. For example, a PCI Express bus [12] can transfer data three orders of magnitude faster than the real-time SAFEbus [7].

However, the main challenge when integrating COTS peripherals within a real-time system is the unpredictable timing of the I/O subsystem since COTS components are typically designed paying little or no attention to worst-case timing behaviors. In particular, we are concerned with I/O subsystems with high bandwidth requirements; a modern real-time system such as a search and rescue helicopter [16] may include several high-bandwidth components such as a Doppler navigation system, a forward look-ahead infrared radar, a night vision system, and several types of communication systems. Modern I/O components such as these can inject significant traffic onto the I/O bus. For example, a single real-time high-definition video may consume an I/O bandwidth of tens to hundreds of Mbps [1].

While priority-based real-time scheduling is a standard practice for the CPU, it is currently not supported by COTS peripherals and interconnect systems (e.g., PCI bus [12]). Due to the lack of real-time prioritization, data I/O transactions travelling through the COTS bus into or out of main memory can suffer unpredictable delay and cause deadline misses [11]. Unfortunately, end-to-end real-time guarantees can not be achieved unless both tasks and I/O data transactions are properly processed in a prioritized manner. We address this challenge by introducing a real-time I/O management system that supports a wide range of priority-based scheduling policies, retains backward compatibility with existing COTS-based components, and achieves high real-time bus utilization without degrading peripherals' throughput. The proposed framework acts like a "transparent layer" that does not add any additional burden at the operating system or user level, except for assigning a certain priority to each real-time I/O flow. In light of the discussed problems, the main contributions of this paper are:

- the design of a real-time I/O management system that supports (in a transparent and backward compatible manner) a wide range of priority-based scheduling policies for COTS interconnect components;
- an extension of Real-Time Calculus theory [3] to determine the I/O delay bounds and each bridge's necessary buffer size to guarantee lossless I/O traffic delivery;
- a working prototype whose experimental measurements validate the effectiveness of proposed real-time I/O management system.

In earlier work, Pellizzoni *et al.* proposed a coscheduling framework between CPU and I/O peripherals to guarantee main memory latency for tasks running on the CPU [13,14]. The authors proposed to use passive Peripheral gate (P-gate) devices to block and unblock peripherals (possibly reducing I/O throughput and causing internal peripheral buffers to

overflow) and to synchronize I/O activity with tasks executing on the CPU. Although this technique was effective for predicting each task's worst-case execution time (WCET) in spite of I/O traffic spikes, it did not guarantee the timeliness of I/O traffic. To the best of our knowledge, the proposed real-time I/O management system provides significant advancement over the current state of the art. In fact, while it remains compatible with the cited timing analysis to guarantee main memory latency for tasks running on the CPU, the real-time I/O management system provides the following novel features: 1) it enforces predictable bandwidth reservations for I/O COTS peripherals, 2) it does not require synchronization between the CPU scheduler and the I/O subsystem, and 3) it facilitates lossless reshaping (under given assumptions) of bursty traffic from a network of distributed real-time nodes. Finally, it is worth noticing that the proposed real-time I/O management system is completely transparent to main CPU applications: in fact, we were able to use a network card through our real-time I/O management system prototype without any user-end application modifications.

The paper is organized as follows. First, in Section 2, we elaborate on the design of the proposed real-time I/O management system. This system, transparent to end-user CPU applications, introduces two new types of components into the COTS system, real-time bridges and a reservation controller, to provide temporal isolation on the COTS bus. In Section 3, we elaborate on our Real-Time Calculus based delay and buffer size analysis. We then demonstrate, on physical hardware, that timing violations can occur without our real-time I/O management system, but with our system we can prevent I/O deadline misses in Section 4. We finish with related work in Section 5 and then conclusions and future directions in Section 6.

2. Real-Time I/O Management System Design

Before describing our real-time I/O management system for predictable I/O performance on a COTS system, we first describe the way in which a COTS system typically works. A COTS system may include several commercial peripherals, such as video acquisition boards or network cards, plugged into standard buses, such as PCI or PCIe, on a commercial motherboard. Data from these boards would travel through a series of bridges and buses (the specifics depend on the model of the motherboard), until it reaches main memory, where the CPU can read it through the Front Side Bus (FSB). Alternatively, the CPU could write data into main memory and instruct the COTS peripherals to retrieve it. For example, a network card could be instructed to upload packets which are stored in RAM.

Our proposed real-time I/O management system, shown in context in Figure 1, adds two types of components to the existing COTS system. The first type is a reservation controller, which implements the system-wide policy for accessing the bus. It can be thought of as a high-level arbiter

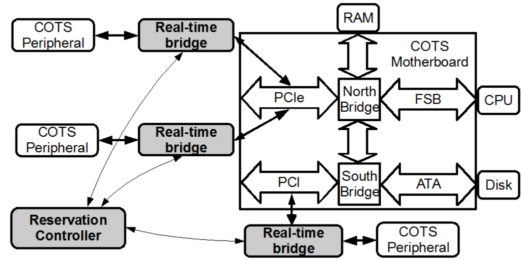

Figure 1: The proposed real-time I/O management system adds a reservation controller and real-time bridges to the COTS-based node.

which instructs the real-time bridges to either communicate on the bus, or yield to other devices. The other type of component we introduce is a real-time bridge which is interposed between each peripheral and the communication bus. Each real-time bridge provides the actuation mechanism to enforce peripheral bus access. For rapid development, we implemented both of these components in hardware on field programmable gate arrays (FPGAs), although an industrial application would likely use an Application Specific Integrated Circuit (ASIC). We describe these devices in Section 2.1 and Section 2.2 and elaborate on details involved in making the prototypes.

2.1. Reservation Controller

Multiple peripherals must cooperate to prevent a timeliness reduction caused by mutual interference. The reservation controller centralizes decision making and coordinates multiple real-time bridges by instructing them to either forward or buffer peripheral traffic. Presently, we consider each peripheral as generating a single real-time I/O flow. In Section 2.2, we describe the changes necessary to permit a single peripheral to send multiple real-time I/O flows with potentially different priorities. Furthermore, we only consider I/O flows directed to main memory, and not to other peripherals[1]. As described in Section 1, and shown later in our evaluation in Section 4, allowing multiple COTS peripherals to simultaneously access main memory can result in unpredictable bandwidth allocation. As such, we allow only a single real-time bridge to transmit at any one time. Therefore, we can consider the time allocated among all real-time bridges by the reservation controller as a shared resource akin to a monoprocessor CPU. In this analogy, each I/O flow is equivalent to a real-time task, and each I/O data chunk in the flow is equivalent to a job; transfer times for I/O data chunks are equated to computation times and can typically be derived by dividing the I/O data amount by the achievable throughput of the real-time bridge.

Coordination between the reservation controller and each real-time bridge is achieved using two physical wires. Each real-time bridge communicates one boolean value,

1 Our methodology will be extended to cover inter-peripheral communication in our future work.

Code 1 These logical expressions, implemented in hardware on the reservation controller, provide a static-priority I/O scheduler for a four-peripheral system.

$\texttt{block}_0 := \neg(\texttt{data_rdy}_0)$

$\texttt{block}_1 := \neg(\texttt{block}_0 \wedge \texttt{data_rdy}_1)$

$\texttt{block}_2 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{data_rdy}_2)$

$\texttt{block}_3 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{block}_2 \wedge \texttt{data_rdy}_3)$

`data_rdy`, to the reservation controller. This value indicates that data is buffered and ready to be sent on the bus. In turn, the reservation controller sends one boolean value back to the real-time bridge, `block`, which instructs the bridge to either block I/O traffic or permit bus access. Real-time bridges instructed to block do not attempt to gain access to the bus, mandating the bus arbiter grants the unblocked peripherals access to the bus to send their data.

With only these two signals, many kinds of bus scheduling policies can be enforced. Consider, for example, scheduling four real-time bridges according to a static-priority bus scheduling scheme, such as rate monotonic (RM). Let $\texttt{block}_i$ be the block command sent to the ith real-time bridge, and let $\texttt{data_rdy}_i$ be the indicator of buffered data coming from the ith real-time bridge. Let the bridges be physically connected to the reservation controller in the order of their priorities (in the order of their rates for RM), from the highest priority bridge, $i = 0$, to the lowest priority bridge, $i = 3$. In order to provide static-priority scheduling on the I/O bus, the reservation controller hardware would implement the logical expressions in Code Block 1.

Our framework, however, can also support a large class of monoprocessor scheduling algorithms which handle sporadic and aperiodic tasks using real-time servers [4]. In our prototype, for instance, we have implemented support for sporadic servers [2] under fixed-priority scheduling (see Section 2.1.1). Notice that the servers are implemented on the reservation controller and not on the associated real-time bridges. This decision has two major advantages. First, it removes the need for precise clock synchronization among real-time bridges and the reservation controller. Since all scheduling servers use the same physical clock, server budgets can be precisely calculated without clock skew. Second, it simplifies the interface between each real-time bridge and the reservation controller by reducing the number of physical wires to just two, `data_rdy` and `block`. This becomes a concern for algorithms like EDF, where each server must communicate a precise deadline timestamp to the scheduling algorithm, which requires dozens of bits of information. By centralizing all scheduling servers on the reservation controller, the number of physical wires is reduced, simplifying the electrical design.

We now describe the way in which our proposed architecture can be used to provide real-time guarantees for peripheral traffic. Particularly, we are interested in determining the classes of schedulers that can be implemented using multiple real-time bridges connected to a reservation controller. We address the following question: can any real-time monoprocessor scheduling algorithm be used with our hardware framework? Although we show that the answer is no, we are still able to employ a large class of monoprocessor schedulers. Since there are many classifications of scheduling algorithms, we start by showing a definition that provides a more formal framework on which to reason about implementable schedulers.

Definition 1 (Scheduling Servers / Scheduling Algorithms)**.** *Consider a set of tasks requesting access to a single shared resource. Each task (or group of tasks) has an associated* scheduling server *which, based on the task's (tasks') activity, forwards scheduling parameters to the* scheduling algorithm. *At any time instant t, the scheduling algorithm grants the shared resource to at most one scheduling server based on the value of all received scheduling parameters.*

The provided definition is general enough to encompass all common real-time schedulers for a single shared resource. For example, consider scheduling periodic tasks according to rate monotonic. The scheduling server for each task forwards a STATIC_PRIORITY parameter equal to the inverse of the task period, and a READY boolean parameter which is true if and only if the task has remaining computation time. The fixed priority scheduling algorithm then selects the task with highest STATIC_PRIORITY and a true READY parameter to execute. A budget-based server for aperiodic tasks, such as a sporadic server, would be more complex, but would still output the same STATIC_PRIORITY and READY parameters. In particular, the server's READY parameter is true if and only if the served task has remaining execution and there is available budget. However, the set of scheduling parameters changes based on the scheduling algorithm. For instance, an EDF server would need to output both a READY boolean value and a dynamic ABSOLUTE_DEADLINE parameter.

The logical division between the scheduling servers and the scheduling algorithm corresponds to the physical implementation within the reservation controller. Each scheduling server is implemented by a single VHDL hardware module. The module receives the `data_rdy` signal from the corresponding Real-Time Bridge and computes the scheduling parameters (the boolean READY value and the STATIC_PRIORITY value). Global scheduling logic, such as the logic shown in Code Block 1, then implements the scheduling algorithm, receiving scheduling parameters and producing `block` signals for all scheduling servers (however, in the case of fixed priority scheduling, STATIC_PRIORITY is a design-time parameter which is absorbed in the implementation of the global scheduling logic as an optimization).

The main drawback of such a design, however, is that each scheduling server must make scheduling decisions based only on the task's `data_rdy` (active) value.

Definition 2 (Active-Dynamic Server)**.** *A scheduling server is an* active-dynamic server *if run-time task behavior can be governed using only one piece of task-related knowledge, whether the task is active (immediately executable) or not.*

Proposition 1. *Our I/O scheduling mechanism can implement any scheduling framework where all scheduling servers are active-dynamic servers.*

Next, we provide some examples of servers that are active-dynamic, and some examples of servers that are not.

Lemma 1. *Under fixed priority scheduling, both a periodic task server and the sporadic server are active-dynamic servers.*

Proof. A scheduling server servicing a periodic task needs only output the STATIC_PRIORITY for the task, and a dynamic READY parameter which is equal to the active value of the task.

A sporadic server is active-dynamic because the rules governing its replenishment time and replenishment amount can be obeyed knowing only the task's active value (and other non-task parameters such as the current time). According to the rules of a sporadic server, the replenishment time is determined as soon as the task becomes active (an aperiodic task requests execution) and there is available budget. The replenishment time is computed by simply summing the current time with the static server period. The replenishment amount, computed when the server becomes inactive, is equal to the consumed budget. This is computed by measuring the duration of time the task was both active and unblocked (which can be given through feedback from the scheduling algorithm). Determining when the server becomes inactive is done by checking if the task is finished or the budget is exhausted. Since all the server rules can be evaluated based only on the active status of the task, a sporadic server is an active-dynamic server. □

Lemma 2. *Consider a sporadic task feasibly scheduled under EDF with relative deadline greater than its interarrival time. The server for such task is not an active-dynamic server.*

Proof. Consider two successive jobs j and $j+1$ of the sporadic task. Since the interarrival time between jobs is less than the relative deadline, job $j+1$ could arrive in the system before job j is finished. Since the server has no way to know when $j+1$ arrived based on the active status of the task, it can not set the deadline for job $j+1$. □

Also notice that there exist non budget-based servers, such as the total bandwidth server [22], which require arrival-time information about aperiodic job execution time and therefore are not active-dynamic servers.

Two additional architectural attributes need to be considered when developing the proposed real-time I/O management system. First, the I/O transfer time is only valid if the device receives the achievable throughput on the front side bus (FSB). Since the CPU main memory access is not regulated by the reservation controller, it may concurrently access main memory. Second, for performance reasons, typical COTS buses do not allow individual bus transactions to be preempted and so they must run always to completion. This means that although we may activate a real-time bridge's `block` signal, the bridge will still need to complete its current transaction before relinquishing control of the bus, which may affect the schedulability. We address these concerns in order.

The first concern is that there may still be contention on main memory even if all other peripherals do not transmit on the bus. The CPU may concurrently attempt to access main memory. From a general framework perspective, we would need to pessimistically account for any increase in bus transfer time due to uncontrolled CPU interference through a technique similar to that used to analyze the reverse problem, CPU main memory delay because of peripheral interference [14]. However, there is a key difference which makes the analysis easier, in that PCIe and PCI transactions are typically buffered within the interconnect. This means that unless the main memory bandwidth is exceeded, the peripheral will not notice any increase in transfer time because of the CPU accessing main memory (the reverse, however, is not true because the interconnect does not typically buffer CPU main memory access). In a typical COTS architecture, the cumulative bandwidth consumed by the CPU and each individual high-speed peripheral does not exceed the bandwidth of main memory, so calculating the I/O transfer time is straightforward.

The next concern is that, since individual bus transactions must be allowed to run to completion because of the COTS bus, the reservation controller's `block` command is not acted upon immediately. In order to evaluate the impact of this delay, we computed its maximum value. A typical single lane PCIe device has an achievable bus throughput of 250MB/sec where each bus transaction is 4KB. This results in single bus transaction time of 16 microseconds. Since this is three orders of magnitude less than our I/O period (a high resolution video stream at 50 frames per second has an I/O period of 20ms), we consider its effect negligible for the schedulers we evaluate. However, if the reservation controller uses a scheduler that switches between devices with the same granularity as the single bus transaction time (for example a least slack first scheduler), then this effect would need to be taken into account (for example by adding an `executing` output from each the real-time bridge to measure exact bus access time).

2.1.1. Prototype Details In our prototype, the reservation controller is built using the Xilinx ML505 Evaluation Platform [26] which features an XC5VLX50T FPGA. We created VHDL hardware code to implement the rate monotonic scheduling algorithm [10] for strictly periodic tasks, as well the sporadic server algorithm [20] to schedule aperi-

odic peripheral bus traffic. The implementation itself is split into two types of hardware components: 1) servers for each real-time bridge, and 2) a global scheduling algorithm. Each server generates a READY signal, and the global scheduling algorithm in this case executes the task with the highest static priority and an asserted READY signal.

In terms of FPGA area utilization, our implementation is lightweight. We synthesized a two layer reservation controller with four sporadic servers scheduled according to a global RM scheduling policy. The resultant implementation used 531 of the 28,800 available slice registers (1.8%), and 668 of the available 28,800 slice look up tables (LUTs) (2.3%). The number I/O buffers used, which is associated with the chip's pinout, was also low (9 out of 480, 1.9%). These numbers lead us to conclude that the design will easily scale to concurrently manage dozens of I/O peripherals.

2.2. Real-time Bridge

In order to provide real-time guarantees on bus communication, bus access must be controlled according to the policy dictated by the reservation controller. Since off-the-shelf peripherals are unlikely to have such a mechanism built-in, we interpose a device between each peripheral and the bus in order to provide this functionality to our real-time I/O management system. In addition to restricting bus access, *real-time bridges* also provide an important additional service to connected peripherals. Each real-time bridge provides a buffer which is able to store pending traffic while bus access is prohibited. This allows high-bandwidth peripherals to be blocked from the bus for relatively long periods of time without suffering from data loss due to full internal buffers on the COTS peripheral. Combined with a communication guarantee provided by the reservation controller's scheduling policy, this guarantees the I/O system will deliver all communication traffic by its I/O deadline.

2.2.1. Prototype Details We envision a final general real-time bridge using a setup similar to the ML455 [28], an FPGA development platform which can be directly interposed between a COTS bus and a COTS peripheral. This device contains both a PCI-X edge connector and a PCI-X socket slot connected to the same Virtex 4 FPGA chip, which would allow various types of peripherals to use the exact same real-time bridge. However, in order to rapidly develop a complete prototype, we focused our effort on a real-time bridge for one specific peripheral. We targeted a network interface card on the ML505 FPGA Evaluation Platform [26]. This device features both an Ethernet hardware interface as well as a one-lane PCIe edge connector both connected to the same Virtex 5 FPGA chip. We first describe the hardware components of the System-on-Chip (SoC) design, and then elaborate on the software development effort.

Figure 2: Our real-time bridge prototype is a System-on-Chip implemented on the ML505 FPGA Evaluation Platform.

Hardware Design. A logical outline of the important hardware components in the network interface card version of the real-time bridge is shown in Figure 2. We now describe each of these in the order of the dataflow through the real-time bridge during normal operation. Consider the case where a packet arrives through the Ethernet connection.

First, the physical hardware interacts with the **Tri-state Ethernet MAC (TEMAC)** hardware block. This is a fixed hardware block on the FPGA, and is the COTS peripheral that the real-time bridge is managing. This block maintains a set of memory addresses where to place packets after they are received. After the packet arrives to the TEMAC, it gets stored into the **FPGA DRAM** and an interrupt is raised to the **Microblaze Soft CPU** [27] which provides information on where the packet was stored and the size of the received data. The Microblaze processor is a soft CPU, meaning that is implemented using the reconfigurable logic on the FPGA. We developed a driver running on the Microblaze that will take the addresses and lengths of the packets and put them into a download queue of data to be sent to the main system. This queue exists in two parts. The potentially long tail of the queue is stored in software, whereas a bounded number of entries (say 128) of the front of the queue are stored in hardware on the **Bridge DMA Engine**. The Bridge DMA Engine, which is a hardware block we created specifically for the real-time bridge, manages actually moving the data out of FPGA DRAM and transferring it into the host CPU's main memory. Along with the queue of data needing to be transferred, the Bridge DMA Engine manages the `block` and `data_rdy` signals on the real-time bridge. When `block` is asserted, the Bridge DMA Engine will not transfer data out of FPGA memory. The `data_rdy` signal is asserted whenever any data is in the hardware queue. The DMA transfers themselves are abstracted as an address to address copy on the Processor Local Bus (PLB). The **PLB / PCIe Bridge** handles the process of translating a write transaction on the PLB bus to a write transaction on the PCIe bus. When the Bridge DMA Engine is unblocked by the reservation controller and performs a DMA operation, the memory containing the packet in FPGA DRAM is copied into the **Host DRAM**. After the transaction is complete, an interrupt is raised on the **Host CPU**, which then takes the

packet data and passes it to the network stack for processing.

Sending packets out from the main CPU works in a similar way, but in the reverse order. The main CPU stores the packet data in Host DRAM and then writes the addresses to an upload queue which resides both in software as well as on the Bridge DMA Engine. When the Bridge DMA Engine is unblocked, it transfers the packets from Host RAM into FPGA DRAM (the PCI / PLB Bridge will again do address translation) and raises an interrupt to the Microblaze Soft Processor. Our driver on the Microblaze then sends the TEMAC hardware block the addresses and lengths of the packet data in FPGA DRAM. Finally, the TEMAC hardware sends the data over the physical Ethernet medium.

An important detail of this implementation is that the (comparatively) slow Microblaze processor does minimal processing (working with only the addresses and lengths) and no copying of the potentially high-bandwidth packet data. This allows our prototype implementation to achieve a network throughput of about 100 Mbps for upload and 80Mbps for download, which coincides exactly with Xilinx's TEMAC performance benchmarks for our setup (ML505, 125MHz Microblaze, 1500 Maximum Transmission Unit (MTU)) [6]. Performance can be further improved by using a larger MTU. Additionally, without the Microblaze bottleneck, the Bridge DMA Engine was able to send data at 207MBps, which approaches the theoretical limit of a PCIe single lane connection (250MBps).

It is worth noticing that the proposed real-time I/O management system introduces additional latency compared with a COTS peripheral communicating directly to the PCIe bus. This is one tradeoff that is made in order to provide control of peripheral bus access. We ran an experiment to get an idea of the effect of this additional latency by sending ping packets to the main CPU through our real-time I/O management system (with the real-time bridge scheduled as the highest-priority sporadic server) and sending ping packets directly to the FGPA's PetaLinux OS. Surprisingly, the packet round-trip times through our bus scheduling prototype (2.40ms) were actually *lower* than the round-trip times for the packets processed immediately on the FPGA (2.62ms). Hence, it is faster to place the packet data in the Bridge DMA Server, assert `data_rdy` to the reservation controller, wait for the `block` signal to be deasserted, receive access to the PCIe bus, transmit the data into Host DRAM, process the ping on the host's 2.66 GHz CPU, and reverse the entire process for the ping response, than to handle the ping packet directly on the slower 125 MHz Microblaze processor. This experiment demonstrates the efficiency of our implementation.

This overall hardware framework currently supports a single real-time flow through each real-time bridge. In order to allow multiple real-time flows through a single real-time bridge, for example to distinguish different TCP connections on the same network interface, which vary in real-time importance, we would need to replicate the Bridge DMA Engine for each flow we would like to support. Since the scheduling wires interact only with the Bridge DMA Engine, the reservation controller would not need to be modified. However, in such a design the FPGA Microblaze processor or a custom hardware block would need to read some packet data, for example the port, to distinguish between real-time flows and forward the address/length information to the corresponding Bridge DMA Engine, which may add overhead.

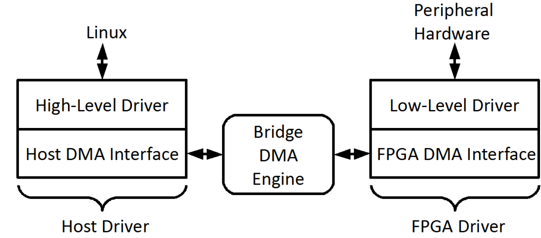

Figure 3: The real-time bridge software architecture consists of a host driver and an FPGA driver.

Software Design. The software architecture for the real-time bridge is paramount to our real-time I/O management system's applicability. Although the hardware design may be generalized for a particular bus interface and therefore reused, each unique COTS peripheral requires some software effort to be transparently controlled by a real-time bridge. In particular, each peripheral requires two drivers, a *host driver* to run on the main CPU, and an *FPGA driver* to run on the real-time bridge's Microblaze Soft CPU. A logical layout of their interactions is show in Figure 3.

One advantage of using COTS peripherals is that it is common for stable, open-source Linux drivers to already exist for peripherals that we are interested in controlling, which simplifies the software effort. Thus, in order to maximize code reuse, we run the Linux kernel on both the real-time bridge and the host CPU. The real-time bridge's FPGA runs PetaLinux version 0.30rc1 [15], a Linux port for the Microblaze Soft CPU based on version 2.6.20 of the Linux kernel. The host system runs a Linux kernel, version 2.6.29. Importantly, the driver on the main system is designed to make the real-time bridge completely transparent to the end user and end-user applications. In Figure 3, the Host DMA Interface portion of the host driver and the FPGA DMA Interface portion of the FPGA driver can be reused in all real-time bridges. The high-level and low-level driver portions can be extracted from an open-source Linux driver for the targeted COTS peripheral. We now elaborate on each of the drivers.

The **host driver**, which is a kernel module running on the main CPU, creates a network card interface for the real-time bridge. From the user's perspective, there is no difference between using a network card directly and using a network card through a real-time bridge.

To handle incoming traffic from the network, the host driver allocates memory for incoming packets and maintains the *available addresses hardware queue* on the Bridge DMA Engine. When a packet arrives, the Bridge DMA Engine uses these addresses to store packet data and then may raise an interrupt. To reduce overhead, the Bridge DMA Engine does not raise an interrupt for every packet transferred. Instead an interrupt is raised when any of these conditions becomes true:

- one or more packets have been transferred, and the Bridge DMA Engine has no more packets ready to be transferred,
- one or more packets have been transferred, and the `block` line becomes asserted,
- the number of transferred packets since the last interrupt reaches a design-time parameter.

Every time an interrupt is raised, the driver, without making a copy, delivers the packet addresses and lengths to the Linux network layer. The Linux network layer then processes and frees the packets. Finally, the driver refills the Bridge DMA Engine's *available addresses hardware queue* with new addresses to replace those that were consumed.

For outgoing traffic, the driver receives packets from the Linux network layer and puts their lengths and addresses into the *outgoing packet hardware queue* on the Bridge DMA Engine. If the *outgoing packet hardware queue* fills up, the driver blocks the network layer until some packets have been transferred. The network layer, in turn, stores the backlogged packet information in a software queue. After the packet data is transferred to FPGA DRAM, an interrupt is raised by the Bridge DMA Server using the same rules as incoming packets, to inform the driver that the memory associated with the transferred packets can be freed.

The only dependence of the host driver on the COTS peripheral is the type of interface that is exported to Linux. This means that, if we were to design a real-time bridge for a different type of network interface card, the host driver would remain the same. To adapt the driver for a different class of peripherals, only the Linux interface would need be changed[2]. In terms of driver reuse, the Linux interface is contained in the high-level driver. Even if a Linux driver does not exist for the specific COTS peripheral we are using, all drivers for the same type of peripheral will contain the same Linux interface and therefore the same high-level driver. Thus, for developing the host driver, any existing driver for the same type of peripheral that we are using will reduce software effort.

The **FPGA Driver** runs on the Petalinux kernel on the real-time bridge. The driver consists of an FPGA DMA Interface, and the low-level driver for the TEMAC hardware block. The driver deals only with lengths and addresses of packets, and does not make copies or perform any processing in order to minimize the performance impact of the slow 125MHz Microblaze Processor.

Incoming and outgoing packets work as expected, with a few intricacies. When incoming packets become queued, perhaps because the `block` signal is asserted to the real-time bridge, the *incoming packet hardware queue* may become full. In this case, the driver maintains a software queue to store incoming packets until space becomes available in the *incoming packet hardware queue*. Therefore, packet drops and retransmissions are avoided resulting in better performance of network protocols like TCP. Additionally, the FPGA driver shares information with the Host Driver in order to propagate network parameters such as the MAC address and the Maximum Transmission Unit (MTU).

Even though the FPGA driver has direct interaction with the COTS hardware, most of the code can be reused from either the already-developed FPGA DMA interface, or the low-level driver portion of an existing Linux driver. For example, in our network card real-time bridge prototype, we use the existing TEMAC driver until the packets are ready to be given to the Linux network stack and then instead instruct the Bridge DMA Engine to send them to host memory. For outgoing packets, the data received from the Bridge DMA Engine is sent to the TEMAC hardware block using the same interface that the PetaLinux network layer uses.

In the worst case, if no driver source code is available for the COTS peripheral we want to control, the development effort required to develop the entire driver is still strictly less than the non-COTS approach: building application specific hardware in addition to the entire driver.

3. Formal Flow Analysis

The main advantage of the equivalence between I/O and CPU scheduling described in Section 2.1 is that feasibility can be checked using any suitable monoprocessor schedulability test, given either an I/O period and deadline information (for a periodic flow) or real-time server parameters (for an aperiodic flow). However, checking I/O feasibility is not enough to ensure system-level correctness. Data traffic must be processed by computational tasks, hence, in a distributed application, communication delay can significantly impact the overall end-to-end application delay. Furthermore, we must ensure that each real-time bridge has enough memory resources to buffer all the incoming data before it is transmitted on the bus, and similarly, that enough space is available in main memory to buffer all the outgoing data before it is fetched by the Bridge DMA Engine.

Delay and buffer space computation are easy for a periodic or sporadic flow. Techniques to compute worst case response time are available for both fixed priority [9] and EDF [21]. Let e be the maximum job size in bytes, p the period or interarrival time of the task, and r its response time. Then the maximum flow delay is simply r and the required buffer size is $\lceil \frac{r}{p} \rceil e$. However, in the case of an ape-

2 The Linux interface determines the class of the device (character device, block device, or network interface), and the relevant device file operations.

riodic flow serviced by a budget-based server, the analysis is not so trivial. Note that incoming network flows in a distributed system are rarely exactly periodic: even if they are generated by periodic tasks and transmitted according to a highly predictable scheduling scheme such as TDMA, they likely accumulate jitter as they traverse multiple bridging and routing elements. Furthermore, clock skews are typically present due to lack of accurate timing synchronization in large distributed real-time systems.

To address this problem, we propose an analysis methodology for fixed priority scheduling based on the theory of Real-Time Calculus [23]. Real-Time Calculus extends the basic concepts of Network Calculus [3] and provides delay and buffer bounds for deterministic data and event flows. Network Calculus has been applied to model a variety of networking and routing scenarios, and is therefore well suited to provide characterization for incoming network data flows.

In Real-Time Calculus, a flow trace is described by a cumulative function $R(t)$, which represents the resource amount required by the flow in the interval $[0, t]$. A representation of all possible traces for the flow is provided by a tuple $\alpha(t) = \{\alpha^u(t), \alpha^l(t)\}$ of upper and lower arrival curves: $\alpha^u(t)$ is an upper bound on the resource requirement in any interval of length t, while $\alpha^l(t)$ is a lower bound on the resource requirement in the same interval. Formally:

$$\forall R, \forall s < t : \alpha^l(t-s) \le R(t) - R(s) \le \alpha^u(t-s). \quad (1)$$

Similar to arrival curves, a representation of the availability of a shared resource can be described by a tuple $\beta(t) = \{\beta^u(t), \beta^l(t)\}$ of upper and lower service curves: $\beta^u(t)$ represents an upper bound on the amount of service time provided to a backlogged flow during any interval of length t, while $\beta^l(t)$ represents the corresponding lower bound[3].

Physical resources are modeled by a collection of abstract components. Each abstract component receives an input flow $\alpha(t)$ and available resources $\beta(t)$, and produces an output flow $\alpha'(t)$ and service curve $\beta'(t)$ representing the remaining resources. The relations among input and output arrival and service curves depend on the processing semantics of the modeled flow and resource. For example, consider a flow that greedily consumes all offered resources when active. The corresponding abstract component can be described by the following relations [5]:

$$\begin{aligned} \alpha'^u &= \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\}, & (2)\\ \alpha'^l &= \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\}, & (3)\\ \beta'^u(t) &= \inf_{\lambda \ge 0}\{\beta^u(t+\lambda) - \alpha^l(t+\lambda)\}, & (4)\\ \beta'^l(t) &= \sup_{0 \le \lambda \le t}\{\beta^l(t-\lambda) - \alpha^u(t-\lambda)\}, & (5) \end{aligned}$$

3 Note that the concept of service curve in Real-Time Calculus is equivalent to that of strict service curve in Network Calculus.

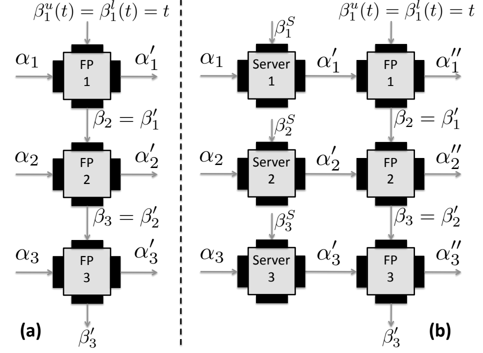

Figure 4: Abstract components are interconnected to model scheduling policies.

where the min-plus convolution $\otimes$ and the min-plus deconvolution $\oslash$ of two functions f and g are defined as:

$$\begin{aligned} (f \otimes g)(t) &= \inf_{0 \le \lambda \le t}\{f(t-\lambda) + g(\lambda)\}, & (6)\\ (f \oslash g)(t) &= \sup_{\lambda \ge t}\{f(t+\lambda) - g(\lambda)\}. & (7) \end{aligned}$$

Abstract components can be composed to model specific scheduling policies. For example, in Figure 4(a) we show how to model fixed priority arbitration for a system with three flows τ_1, τ_2, τ_3 in decreasing order of priority. The abstract component modeling flow τ_1 can use all available resources, the abstract component for τ_2 only gets the resources that were not consumed by τ_1, and so on. Finally, maximum delay $d^{\max}$ and required buffer space $b^{\max}$ for each flow can be computed based on input arrival and service curves as follows:

$$\begin{aligned} d^{\max} &= \sup_{\lambda \ge 0}\{\inf\{t \ge 0 : \alpha^u(\lambda) \le \beta^l(\lambda + t)\}\}, & (8)\\ b^{\max} &= \sup_{\lambda \ge 0}\{\alpha^u(\lambda) - \beta^l(\lambda)\}. & (9) \end{aligned}$$

To account for the effect of scheduling servers, we propose to add an additional abstract component for each flow as shown in Figure 4(b). In our model, each flow is characterized by three arrival curve tuples: α, α' and α''. For received network traffic, α represents the input flow to the real-time bridge, and α'' represents the flow transmitted on the PCI bus. Resource requirements are expressed in terms of transmission time by the real-time bridge. Let C be the achievable transmission speed of the real-time bridge. Then, if a maximum of e bytes can arrive in an interval of length $\bar{t}$, it follows that $\alpha^u(\bar{t}) = \frac{e}{C}$. As a simple example, a strictly periodic input flow with period p and constant job size of e bytes can be modeled with arrival curves $a^u(t) = \lceil \frac{t}{p} \rceil \frac{e}{C}, a^l(t) = \lfloor \frac{t}{p} \rfloor \frac{e}{C}$. Finally, α' is the virtual output flow generated by the abstract server component based on the scheduling server processing semantics. In each interval of length t, $\alpha'^u(t), \alpha'^l(t)$ are upper and lower bounds on the transmission time required by the server. In general, for a budget-based server with budget e_s and period p_s, α'

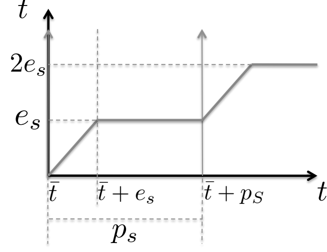

Figure 5: The upper service curve for a sporadic server is derived based on replenishment events (up arrows).

is lower than α, since the server constrains the task to require a transmission time of no more than e_s time units every period p_s.

Next, we determine the way in which to compute service curves β^S for an abstract server component. An end-to-end service curve for each flow can be obtained by concatenating β^{Sl} and β^l computing $\beta^{Sl} \otimes \beta^l$ [3], and delay and buffer bounds can be obtained using the concatenated service curve in Equations 8 and 9.

Theorem 3. *A feasibly scheduled sporadic server with budget e_s and period p_s can be modeled by an abstract component with service curves:*

$$\beta^{Su}(t) = \min\Big(\Big\lceil \frac{t}{p_s} \Big\rceil e_s, t - \Big\lfloor \frac{t}{p_s} \Big\rfloor (p_s - e_s)\Big) \quad (10)$$

$$\beta^{Sl}(t) = \begin{cases} \beta^{Su}\big(t - (p_s - e_s)\big) & \text{if } t \geq p_s - e_s \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

Proof. Consider an interval $[\bar{t}, \bar{t} + t]$ during which the task serviced by the sporadic server is constantly backlogged. Then, an upper bound $\beta^{Su}(t)$ for the service provided by the server to the task in interval $[\bar{t}, \bar{t} + t]$, which is equivalent to the transmission time requested by the server to the fixed-priority scheduler in the same interval, can be obtained assuming that the budget is recharged to the maximum e_s at time $\bar{t}$. This results in the server providing e_s units of service time from $\bar{t}$ to $\bar{t} + e_s$ and again in each interval $[\bar{t} + kp_s, \bar{t} + kp_s + e_s], k \geq 0$. The obtained service curve, shown in Figure 5, is captured by Equation 10.

Now consider the lower service curve $\beta^{Sl}(t)$. Since the task is backlogged and the system is assumed schedulable, the server must provide e_s units of service time to the task every p_s time units. A lower service curve can thus be obtained assuming that the server has no remaining budget for the longest possible interval $p_s - e_s$ before fully recharging it at time $\bar{t} + (p_s - e_s)$. Afterwards the curve is equivalent to the upper service curve, and we can compute it as $\beta^{Sl}(t) = \beta^{Su}\big(t - (p_s - e_s)\big)$. □

While we derived β^S in the case of a sporadic server, the analysis easily extends to other server types by deriving suitable best and worst case arrival patterns.

Board	Data Size	Transfer Time	Budget	Period
ML555	4.0 MB	4.4 ms	5 ms	8 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms

Table 1: Our experiments use four flows.

4. Evaluation

The goal of our evaluation is twofold. First, we demonstrate that there is a problem using COTS interconnect for a real-time system. We present an I/O task set which results in I/O deadline misses when running on a standard COTS bus. Next, we run the same I/O task set within our scheduling framework, and show that all deadlines are met. This demonstrates the non-real-time nature of COTS interconnect, and validates the correctness of our solution.

Performing direct measurements on a high performance COTS I/O system such as PCIe is difficult: the PCIe protocol implements point-to-point connections between each peripheral and the rest of the system running at the very high speed of 2.5 Ghz, making it hard to directly observe. In order to make the most accurate measurements, we used dedicated hardware on the reservation controller. Our trace acquisition hardware module polls the state of the `data_rdy` and `block` signals with a one microsecond resolution. Any changes in these signals, along with an associated timestamp, are output over the reservation controller ML505's serial port where they can be received by an external computer for processing.

We performed experiments on a COTS PC platform with an Intel 975X system controller (northbridge). The selected motherboard has four PCIe slots, allowing us to connect up to four high-speed peripherals. Using a PC platform permits easy access to all PCI slots, however, to derive meaningful measurements, we changed the FSB clock frequency obtaining a theoretical memory bandwidth of 2.4Gbyte/s, which is in line with typical values for embedded platforms.

To make our experiments more easily repeatable, we instructed the smart bridge prototype to generate synthetic traffic instead of using traffic received by the TEMAC over the network. Our periodic task generating drivers run on the main CPU, and since our I/O schedule uses periods on the order of milliseconds, it is difficult to exactly synchronize all synthetic tasks. For this reason, we ran the tests for many hyperperiods, and show here the traces from the most closely aligned arrival times, which correspond to the *near-critical instants*. The arrival times of the presented traces are never separated by more than 0.8 milliseconds. Additionally, we implemented a traffic generator using an ML555 PCI Express Development Board [29] with a faster 8 lane PCIe connection. The synthetic traffic generator is programmed to send a constant amount of data to main memory every period and obeys the I/O scheduling commands from the reservation controller.

The task set used in our experiments consists of four real-

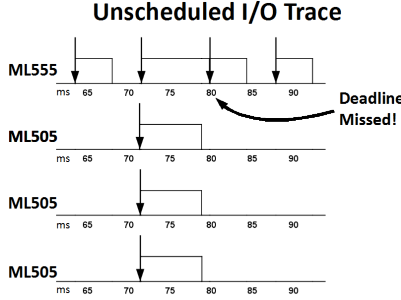

Figure 6: The trace of a standard COTS I/O system reveals a deadline miss if the tasks are released at a near-critical instant.

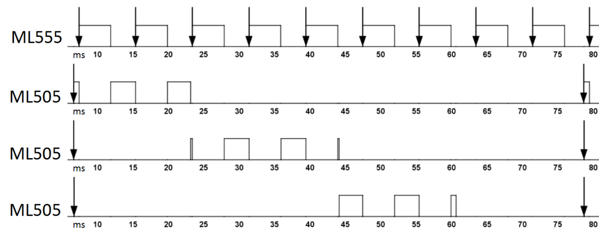

Figure 7: The trace of the task set running with the real-time I/O management shows the system preventing deadline misses by prioritizing traffic.

time flows competing for main memory. The task parameters (data size, transfer time, period) are shown in Table 1. The tasks' periods are harmonic, and the total utilization does not exceed 100%, so the task set is schedulable under RM.

In the first experiment, the COTS bus is used without the reservation controller. Traffic gets sent on the bus as soon as it arrives, increasing the execution time of the ML555's periodic task from 4.4 ms to over 8 ms (an increase of 82%) when the tasks start at a near-critical instant. This causes a deadline miss (see Figure 6). In the second experiment, each peripheral is handled by a sporadic server (whose corresponding budget and period are shown in Table 1) and all the servers are scheduled according to Rate Monotonic with total utilization $\frac{5}{8} + \frac{9}{72} + \frac{9}{72} + \frac{9}{72} = 1$. By using the proposed real-time I/O management system, the task set is now successfully scheduled without missing deadlines because the traffic is prioritized. A trace of one hyperperiod starting at a near-critical instant is shown in Figure 7.

5. Related Work

Apart from our own previous work [13, 14], several papers address the problem of interference at the main memory level. Empirical approaches can estimate the impact of PCI-bus load on task computation time based on experimental measurements of reference tasks [19]. Alternately, analytical approaches exist to bound I/O interference [8]. However, the analysis is restricted to a single DMA controller using predictable cycle-stealing arbitration, and can not be applied to a COTS system. There is also analysis to estimate the impact of mutual interference among processing cores. For example, static analysis can compute cache access delay in a multiprocessor system [17]. However, these results focus on deriving the increase in task execution time while neglecting the effect of delay on communication flows.

Modeling complex COTS interconnections and estimating delay and buffer requirements for peripheral flows can be done in an AADL-based environment [11]. An event-based model may be used to estimate delay for both computation and communication activities in a multicore system-on-chip [18]. However, lack of precise knowledge of COTS behavior implies that these analyses must make pessimistic assumptions, which can lead to high delay and buffer sizes. Our real-time I/O management system removes such unpredictability by forcing an implicit bus schedule.

Several other analysis frameworks exist to estimate delay characteristics for communication flows in embedded systems. For example, Real-Time Calculus can compute end-to-end delay for various real-time systems [23, 25]. Analysis methodologies are available for existing real-time interconnections such as CAN [24]. However, these methodologies typically assume a detailed knowledge of each component's behavior, which is unavailable in a COTS-based system.

6. Conclusions and Future Work

We have presented a framework for providing real-time control of the I/O peripherals in a COTS-based embedded system. This framework involves interposing real-time bridges between COTS peripherals and the COTS interconnect, all of which communicate with a central reservation controller. In this way, we are able to schedule bus transactions such that all I/O deadlines are met, as well as prevent data loss by buffering traffic of high-bandwidth peripherals when bus access is prohibited.

We have shown through experiments that an unmodified COTS I/O system can cause excessive I/O delay leading to deadline misses, while our prototype real-time COTS-based I/O framework provides deterministic delays and meets all I/O deadlines. We demonstrated the way in which classical uniprocessor scheduling theory can be applied within our framework, and created hardware logic to implement the Rate Monotonic and Sporadic Server scheduling policies on COTS peripheral bus traffic. We provided analysis to determine maximum buffer size and delay based on the arrival and service curves of I/O traffic.

One direction for future work is to build upon our current prototype. Currently, each peripheral is considered as a single real-time flow in the system, but this will be expanded to allow different priority, real-time flows to pass through a single peripheral. Additionally, we will provide hardware-based preprocessing and filtering on the real-time bridge, which can be used, for example, to drop less impor-

tant packets if the main CPU is overloaded. Additionally, the real-time bridges can serve as the mechanism for hardware/software co-design of host CPU computations.

Another direction we will pursue is expanding our analysis of the I/O system. Some transactions occur between peripherals which do not involve main memory, and these may be able to be scheduled in parallel without causing interference delay. Additionally, we can think of a scheduling scheme that allows multiple peripherals to transmit at the same time if their interference remains predictable and bounded. Then, we need to avoid multiple potential bottlenecks instead of only main memory bandwidth. For example, PCI uses shared bus segments which can be a bottleneck if multiple I/O transactions co-occur.

Acknowledgement

This material is based upon work supported by Lockheed Martin and NSF under Award No. CNS0720512 and CNS0613665. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF or the supporting companies.

References

[1] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *Proc. of the IEEE International Workload Characterization Symposium*, Oct 2005.

[2] L. Sha B. Sprunt, J.P. Lehoczky. Scheduling sporadic and aperiodic events in a hard real-time system. Technical report, CMU, 1989.

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

[4] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Second Edition*. Kluwer Academic Publishers, Boston, 2004.

[5] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system design. In *Proc. of 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2003.

[6] Doug Gibbs. Measuring treck tcp/ip performance using the xps locallink temac in an embedded processor system. www.xilinx.com/ support/ documentation/ application_notes/ xapp1043.pdf, 2008.

[7] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, March 1993.

[8] Tay-Yi Huang, Jane W. S. Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access i/o concurrent with hard-real-time programs. In *Int. Conf. on Parallel and Distributed Systems*, Tokyo, 1996.

[9] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.

[10] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[11] M. Y. Nam, R. Pellizzoni, R. M. Bradford, and L. Sha. ASIIST: Application specic I/O integration support tool for real-time bus architecture designs. In *Proceedings of the IEEE ICECCS*, Potsdam, Germany, 2009.

[12] PCISIG. Conventional pci 3.0, pci-x 2.0 and pci-e 2.0 specifications. www.pcisig.com, 2009.

[13] R. Pellizzoni, B.D. Bui, M. Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231, 30 2008-Dec. 3 2008.

[14] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time cots based systems. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*, pages 73–82, Washington, DC, USA, 2007.

[15] PetaLogix. Petalinux. http://developer.petalogix.com/, 2008.

[16] John Pike. Hh-60g pave hawk. www.globalsecurity.org/ military/ systems/ aircraft/ hh-60g.htm, 2009.

[17] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28th *IEEE Real-Time System Symposium*, December 2007.

[18] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th CODES+ISSS*, Oct 2008.

[19] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[20] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[21] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, France, January 1996.

[22] M. Spuri and G.C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11, Dec 1994.

[23] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2000.

[24] K. W. Tindell, H. Hansson, and A. J. Wellings. Analysing real-time communication: controller area network (CAN). In *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[25] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*, 9(6):649–667, Nov 2006.

[26] Xilinx. Virtex-5 LXT FPGA ML505 Evaluation Platform. www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm, 2008.

[27] Xilinx. Microblaze soft processor core. www.xilinx.com/ tools/microblaze.htm, 2009.

[28] Xilinx. Ml455. www.xilinx.com/support/documentation/ ml455.htm, 2009.

[29] Xilinx. Virtex-5 LXT ML555 FPGA Development Kit for PCI Express, PCI-X, and PCI Interfaces. www.xilinx.com/ products/ devkits/ HW-V5-ML555-G.htm, 2009.