A Reactive and Cycle-True IP Emulator for MPSoC Exploration

Mahadevan, Shankar; Angiolini, Federico; Sparsø, Jens; Benini, Luca; Madsen, Jan

Published in:
I E E E Transactions on Computer - Aided Design of Integrated Circuits and Systems

Link to article, DOI:
10.1109/TCAD.2007.906990

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

A Reactive and Cycle-True IP Emulator for MPSoC Exploration

Shankar Mahadevan, *Member, IEEE*, Federico Angiolini, *Student Member, IEEE*, Jens Sparsø, *Member, IEEE*, Luca Benini, *Fellow, IEEE*, and Jan Madsen, *Member, IEEE*

***Abstract*—The design of MultiProcessor Systems-on-Chip (MPSoC) emphasizes intellectual-property (IP)-based communication-centric approaches. Therefore, for the optimization of the MPSoC interconnect, the designer must develop traffic models that realistically capture the application behavior as executing on the IP core. In this paper, we introduce a Reactive IP Emulator (RIPE) that enables an effective emulation of the IP-core behavior in multiple environments, including bit- and cycle-true simulation. The RIPE is built as a multithreaded abstract instruction-set processor, and it can generate reactive traffic patterns. We compare the RIPE models with cycle-true functional simulation of complex application behavior (task-synchronization, multitasking, and input/output operations). Our results demonstrate high-accuracy and significant speedups. Furthermore, via a case study, we show the potential use of the RIPE in a design-space-exploration context.**

***Index Terms*—Bus traffic modelling, cycle-true traffic generator, macromodelling, multi-processing, MultiProcessor Systems-on-Chip (MPSoC), network-on-chip, network traffic reproduction, reactive application models, simple instruction set architecture, simulation, systems-on-chip, traffic generator, traffic profiling and trace parsing, traffic shaping.**

I. Introduction

THE primary design paradigm for MultiProcessor Systems-on-Chip (MPSoC) is the separation of the communication and computation concerns, as this enables intellectual-property (IP) reuse and shorter design time. In order to improve the overall performance of an MPSoC platform, it is key to evaluate the impact of the interconnection backbone on the application which is being executed. The interconnect can span over a huge variety of architectures and topologies, ranging from traditional shared buses up to packet-switching Networks-on-Chip (NoC) [9], [13]. Therefore, to select and optimize a particular interconnect, the MPSoC designer needs traffic models that are realistic and accurate. A critical problem is that traffic models should capture not only the behavior of the applications but also that of the applications running on top of a stack of hardware and software; this includes the properties which are not easy to reproduce, such as cache behavior and synchronization.

In the presence of concurrent tasks running on multiple processors, the characterization of traffic patterns is not simply a matter of stochastic modeling [10], [19] or trace-based regeneration [21]. For example, an interprocessor synchronization mechanism based on semaphore polling generates different amounts of traffic depending on the relative timing of accesses. This may create traffic spikes and localized congestion of the interconnect but is very hard to predict in advance, impacting the accuracy of the traffic model. A less simplistic way of modeling the MPSoC system is to describe it entirely as a cycle-true model [15], [22]. This yields the most accurate information for performance analysis and subsequent interconnect optimization. However, the implementation time and the simulation speed of such models are clearly a limit to widespread adoption.

For the purposes of the interconnect designer, a valuable tool for exploration and optimization would be a black-box model that, when plugged at the ports of the interconnect, would act like an IP core, injecting realistic traffic with clock-cycle accuracy. A key desired property would be reactiveness to the surrounding environment, i.e., the ability to adjust traffic patterns depending on synchronization events, which are associated with the system as a whole, and could not be properly rendered by any traffic-generation device in isolation. An example has been given above with synchronization by semaphore polling; more complex scenarios include system-triggered interrupts. Only a tool featuring such reactiveness really allows for meaningful analysis of the interconnect choice and performance. The fundamental problem, however, is how to generate such realistic traffic patterns.

In this paper, we investigate this problem and propose a solution in the form of a Reactive IP Emulator (RIPE) model. RIPE is a tool that can reproduce IP traffic with cycle accuracy. This is done by influencing the type and the timing of the communication transactions based on the current internal state and the synchronization properties of the MPSoC system as a whole. A part of the novelty of our approach is that we use additional and readily available system-level information (such as, for example, the knowledge of the location of semaphore variables in the memory space) to automatically detect synchronization events and respond to them during runtime. These elements allow us to reach the goal of reactiveness.

A first aim of this paper is to investigate the requirements for accurate modeling of communication events on MPSoCs. To do so, in Section II, we will present examples of representative

Manuscript received December 13, 2005; revised August 15, 2006. The work of S. Mahadevan was supported by the ARTIST and the works of F. Angiolini and L. Benini are supported by the Semiconductor Research Corporation (SRC) under Contract 1188. This paper was recommended by Associate Editor A. Raghunathan.

S. Mahadevan, J. Sparsø, and J. Madsen are with the Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark (e-mail: m_shankar@ieee.org).

F. Angiolini and L. Benini are with the Department of Electrical Engineering and Computer Science, University of Bologna, 40136 Bologna, Italy.

Digital Object Identifier 10.1109/TCAD.2007.906990

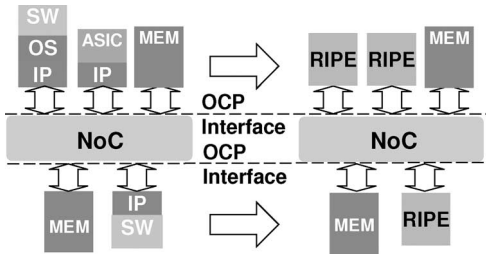


Fig. 1. RIPE as a simulation aid, replacing existing IP cores.

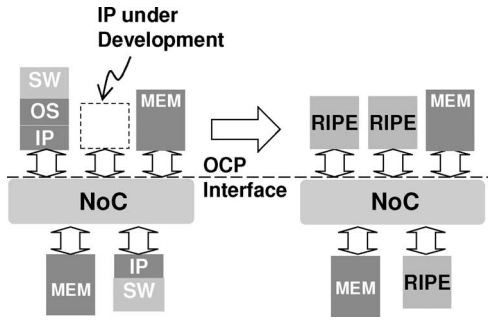


Fig. 2. RIPE as a design tool, acting as an IP core still under development.

applications which are impacted by system-level constraints, such as the sharing of interconnect and memory resources. The scope of this paper will then be extended to systems controlled by an operating system (OS). We will study several interesting cases of usage of interrupts to drive system operation; for example, an interrupt sent by a timer to trigger a context switch may cause a noticeable shift in traffic if the new task has different communication needs.

The key principles of RIPE can be rendered in several types of devices, including behavioral modules, programmable simulation devices, and even programmable hardware blocks. In this paper, we explore the second alternative, which provides the maximum flexibility while leaving future embodiments open. Therefore, we specify an abstract RIPE multithreaded instruction set architecture (ISA), and we build a RIPE SystemC simulation device with Open Core Protocol (OCP) 2.0 [4] sockets at its ports. The RIPE model allows for easy programming of sequences of communication transactions interleaved with idle waits and is capable of sensing and responding to system events and properties. Section III will discuss the details of our implementation and show an example RIPE program which models one of the applications introduced in Section II.

The proposed RIPE device can be used in several ways, which will be discussed in depth in Section IV. One possibility (Fig. 1) is to leverage its features to replace existing IP cores; this usage has previously been introduced in [5] and [23]. The idea is to accurately reproduce communication transactions based on prerecorded system traces. As shown in the figure, by swapping away IP cores for RIPE blocks in the reference cycle-true system, subsequent design-space exploration of the interconnect can be performed independently while keeping a very high level of accuracy and speeding simulation up. A validation scheme for this type of flow will be presented in Section V. Section VI will present the resulting experimental data for a range of complex benchmarks, with and without an underlying OS, when compared against the bit- and cycle-true detailed MultiProcessor ARM (MPARM) [22] model.

On the other hand, a RIPE device can also be used (Fig. 2) in the early stages of the design-space exploration, when all IP cores may not yet have been finalized, in order to explore cosimulation effects and see the impact of hardware changes on the software stack. In this scenario, the interconnect designer may want to leverage RIPE as a design tool by handwriting programs to test specific realistic synchronization-intensive scenarios which would be very difficult to study with traditional traffic-generation flows. For example, in Section VII, we will present a case study where the impact on execution time of variable densities of interrupt events can be investigated.

Finally, we will conclude this paper by contrasting our approach to previous research (Section VIII) and provide conclusions and direction for future work (Section IX).

II. Application Reactiveness in MPSoC Environments

In MPSoC environments, several different types of system-level communication may occur. We identify three broad categories as follows: 1) processor-initiated communication toward a private resource but across a shared medium; 2) processor-initiated communication toward a shared resource; and 3) system-initiated communication toward a processor, which typically happens by means of interrupts. In particular, the second and third types are examples that illustrate the reactiveness of IP cores, a property which must be carefully emulated to accurately model their traffic patterns.

A. Communication With a Private Resource

Let us first consider a simple case of processor-initiated communication toward an exclusively owned slave peripheral but across a shared medium [Fig. 3(a)]. We code a simple application, **"matrix,"** which involves one task per processor, each performing some private computation. No intertask or intercore synchronization is required. However, all tasks compete for access to the same interconnection resource.

In this example, the communication needs of the application are quite easy to model; the result is a simple list of transactions interleaved with computation. The model is made only slightly more complex by the issue of bus congestion, which makes the data-access time unpredictable.

To better understand this issue, please consider the first two master transactions, a write (`WR`) and a read (`RD`). The `WR` transaction can be assumed to be nonblocking for the IP core, which, therefore, simply issues a request and continues its computation. The `RD`, on the other hand, uses blocking semantics. Therefore, the response has to make its way back to the master and only then can computation resume. The overall latency is also a function of the congestion on the interconnect. Therefore, it is not enough to capture a time-annotated list of transactions, as the timing information depends on the specific interconnect. From the emulation point of view, however, a model can be easily achieved as follows. The latency due to

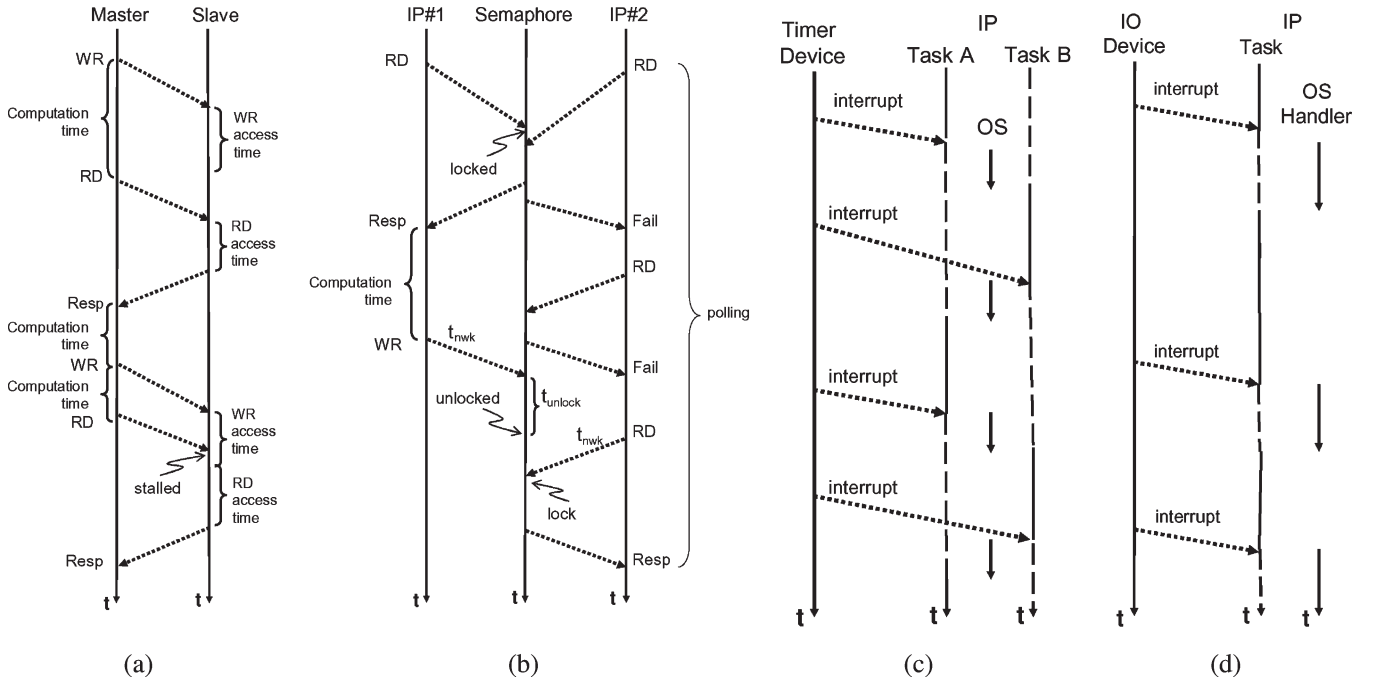


Fig. 3. Timelines for (a) communication with a private memory (**matrix**); (b) communication with a shared memory including polling (**poll**); (c) system-initiated timer interrupt causing a context switch (**multi**); and (d) system-initiated I/O interrupt causing a driver to respond (**IO**).

congestion and actual slave response time can be discarded; the only essential points to capture are just the two transactions: the delay between the WR assertion and the RD assertion (which is computation time), and the delay between the RD response (Resp) and the following command. This information makes it possible to emulate the IP core behavior on any given interconnect, even one having very different latency properties.

Similarly, the stalling behavior observed in the next set of instructions (WR-RD) does not need to be explicitly captured in a RIPE model, since, from a processor perspective, it simply appears to be part of the slave response time.

Requirement #1: This observation leads to the concept of "time-shifting" behavior: consecutive transactions are tied to each other and are issued at times which are a function of the delay elapsed before receiving responses to previous transactions. For emulation purposes, only the length of the computation periods (which can be modeled by idle waits) and the transaction types are needed.

Modeling requirements of this simple category of traffic can be predicted or inferred given an algorithmic specification. In [10] and [31], such an inference is drawn to test the interconnect. However, these models do not hold for more complex traffic types, as those that will be shown in the following paragraphs, unless extremely detailed models of the underlying hardware and software are provided. This includes cache-replacement policies, simultaneous tracking of each processor state, etc.

B. Communication With a Shared Resource

In the simplest synchronization case, one or more processors competing for a shared resource may poll a semaphore to gain resource access. As an example, let us consider a multimedia application called **"poll"** [Fig. 3(b)]. For this case, we map a single task onto each IP core. Tasks are programmed to communicate with each other in a point-to-point producer–consumer fashion; every task acts both as a consumer (for an upstream task) and as a producer (for a downstream task); therefore, logical pipelines can be achieved by instantiating multiple cores. Synchronization is needed in every task to check the availability of input data and of output space before attempting data transfers. To guarantee data integrity, semaphores are provided. A semaphore is a special binary-valued memory-mapped device for which test and set functionality is provided in hardware. Therefore, an RD returns the semaphore state and, if the semaphore is currently "unlocked", also changes its state to "locked." By checking the return value of the RD, the master issuing the command can decide if the locking was successful or if the semaphore had already been locked by another task. The unlocking can be performed with an explicit WR command of the "unlock" value. In the **poll** application, the consumer checks a semaphore before accessing producer output. If the semaphore is found locked upon the first read, the application reacts with a continuous polling strategy, whereby it regularly issues read events until, eventually, the semaphore is found unlocked. Since the transactions occur over a shared interconnect, the unlock event (in this case, the WR issued by IP#1) and the success of the next request (RD event by IP#2) are interdependent.

In the figure, only if the IP#2 RD event is issued at least $t_{\mathrm{nwk,IP\#1}} + t_{\mathrm{unlock,S}} - t_{\mathrm{nwk,IP\#2}}$ after the unlocking by IP#1, then IP#2 will be granted the semaphore, and additional polling events will not be required. Therefore, depending on network properties, a variable amount of transactions might be observed at the ports of the IP cores. This demonstrates that the "time-shifting" behavior introduced before is not sufficient when

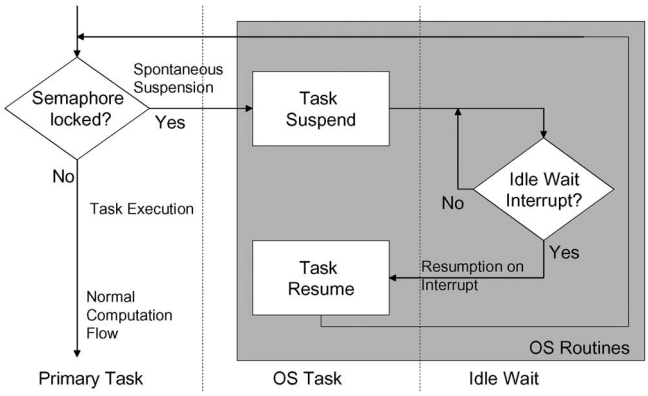


Fig. 4. Application flow of **pipe**.

multimaster systems are taken into account. The arbitration for resources in such designs is timing- and, thus, architecture-dependent.

Requirement #2: The state of the shared resources needs to be tracked. For emulation purposes, the semaphore locations must be known and monitored, and the devices must make use of this information to adjust their execution flows.

C. System-Initiated Communication

System-initiated communication toward a processor is generally performed by means of interrupts, and an OS is in charge of the handling. In reacting to the interrupt, however, the degree of interaction between the OS and the application can vary noticeably. We present here three examples, which are representative of a vast class of execution flows. The RIPE model we will propose can capture all the dynamics of these test cases, given proper insight into the mechanics of the applications and the OS.

As a first example [Fig. 3(c)], we create a test application (**"multi"**) where timer-generated interrupts are used to drive the OS scheduler. In this case, only the OS is aware of the interrupts, while the user tasks are transparently paused and resumed while executing a single stream of operations. In our application, we introduce two tasks per processor, having unbalanced bandwidth needs; therefore, every interrupt causes an abrupt shift in traffic workload for the interconnect.

In a second example [Fig. 3(d)], the **"IO"** application is composed of a main execution task and of a driver for an input/output (I/O) device; the latter is in charge of responding to interrupts sent by the hardware device. The driver operation is bandwidth-intensive, causing traffic spikes on the interconnect.

A third test case (**"pipe"**) features the same logical behavior of the **"poll"** example shown above, which is a pipeline of multimedia processing tasks, but leverages interrupts instead of polling to reduce the congestion on the interconnect and the energy waste upon synchronization points (see Fig. 4). This scenario features a very tight coupling between the application and the interrupt handling; for example, upon an unsuccessful lock acquisition by a consumer, the application interacts with the OS to be descheduled and to be resumed only upon the next interrupt event, which will flag the availability of new data. On the other hand, if the lock is immediately available, the application proceeds directly. Interrupts may be ignored if they are issued ahead of time, i.e., if the notification of new data availability arrives before the consumer is ready to process a new message.

As can be seen, modeling these applications and their impact on interconnect performance is not trivial, presenting a major hazard for any traffic-emulation device. A complete emulation of the hardware and software stacks is needed to properly determine the traffic behavior at the IP core pin-out boundary.

Requirement #3: In presence of interrupt facilities and of an OS, the execution of every application task, of the OS kernel, and of interrupt handlers must be independently identified and modeled. This can be achieved by tracking the occurrence time of interrupt events and application resumptions. The traffic emulator should then be able to model the IP core behavior independently from the interrupt occurrence time.

D. Timing Dependency of Applications

So far, we have evaluated the implications of different MPSoC traffic categories. These requirements are not derived in an *ad hoc* fashion but are representative of typical timing-sensitive real-life applications [33], such as multimedia-stream processing, time-slicing mechanisms in OS schedulers, and I/O device handling. In such applications, the overall performed computation does not change depending on the order of arrival of external events. Therefore, while an execution trace of these examples show widely varying traffic patterns depending on external timings, the major computation blocks are still recognizable. Even though applications with even more timing-dependent behavior do exist, modeling them would require an intratask notion of context switching. At this stage, we believe that the complexity of such an effort for a whole MPSoC, in a generic way, would be excessive and, anyway, unsuited for a black-box component such as the RIPE intends to be.

III. RIPE Model and Implementation

In this section, we describe a particular implementation of the RIPE concept based on an ISA, which is capable of fulfilling the above presented requirements of reactive behavior.

A. RIPE ISA

Applications such as those outlined in Section II can be emulated either within a behavioral/transaction-level module or with an ISA-based device. While our RIPE model and the supporting tool chain (Section V) could also be targeted at the deployment of behavioral models, we choose to develop an ISA-based RIPE implementation, and we describe it in SystemC [2]. While the behavioral model may have a slight advantage in simulation speed over a programmable device, it also requires a recompilation of the simulation platform every time the application to be modeled changes. During design exploration, such a step would be required to study multiple applications on the same platform. A programmable model, with a fixed emulation device and user-written programs, avoids this time-consuming operation, introducing, instead, a simple

TABLE I
RIPE INSTRUCTION SET

Instruction	Description
Communication Instructions:	
`Read(AddrReg)`	Read from an address
`Write(AddrReg, DataReg)`	Write to an address
`BurstRead(AddrReg, CountReg)`	Burst read from address set
`BurstWrite(AddrReg, DataReg, CountReg)`	Burst write to address set
Flow Control Instructions:	
`If(arg1, arg2, operand)`	Branch on condition
`Jump(label)`	Branch direct
`Idle(counter)`	Wait for given no of cycles
`SetRegister(reg, value)`	Set register (load immediate)

TABLE II
RIPE SPECIAL REGISTERS

Special Register	Name	Usage
	Interrupt Registers:	
2	`IntrpMaskReg`	Masks or unmasks interrupts
3	`TaskIDReg`	Stores a task ID
5	`SWIntrpReg`	Sends a software interrupt from within the program
	Other Registers:	
4	`RDReg`	Stores the data value returned by a `Read(AddrReg)` instruction

programming-language paradigm. The designer may not even need to use the language at all when using automatic translation of the traffic specification into a RIPE program, as outlined in Section V. Furthermore, a future goal of our project is to build test chips containing interconnect prototypes. The ISA-based approach is very attractive for this purpose, because it can naturally map onto a hardware device to inject traffic on test chips. In [16], the potential of this type of architecture has already been shown within a field-programmable gate-array (FPGA)-based emulation platform. Compared to current ISA-based emulation and simulation approaches (using tens of instructions), our ISA is simple.

The RIPE is implemented as a nonpipelined processor with a very simple instruction set, as listed in Table I. Its external pin out matches the OCP 2.0 [4] specifications for a master interface. Hardware interrupts are available on the side-band portion (`SInterrupt`) of the OCP interface, and an internal software-interrupt facility is also present. A future planned extension is the support for the multithreading extension of the OCP protocol, thus supporting outstanding and out-of-order transactions. Any other interface standard, such as Advanced eXtensible Interface (AXI) [6], could also be supported, depending on the interface required by the interconnect under study.

The RIPE program that controls the device behavior contains code to model single or multiple tasks. These tasks might be actual tasks running on the IP core which is being emulated or chunks of the OS layer, such as its native interrupt handlers and scheduler. We instantiate in the device a program-counter (`PC`) register, an instruction memory, and a register file for each task specified by the program; no data memory is needed. A context switch among tasks in the task pool is realized simply by referring to the corresponding set of `PC` and register file. The instruction set comprises four instructions for data transfers, whose operation can be controlled by putting proper values in the operand registers. These instructions are blocking, i.e., the RIPE execution is suspended until completion of the OCP handshake, which, for a read, will include the latency of the response over the network.

Four flow-control instructions are also available to realize the reactive behavior. The `SetRegister` instruction loads an immediate 32-b value, which is written into the specified register. The `If` and `Jump` instructions are used to change the execution flow, while the `Idle` instruction models the IP computation periods with idle waits. Within the register file, most registers are general purpose (typically used to set address and data values for OCP transactions), and their number can be configured. Some registers are designated as special purpose. For example, since, in specific-flow control scenarios, the data returned by a `Read` command must be available for evaluation (e.g., in case of semaphore checks), the RIPE device provides in Register 4 the response to the preceding read. Table II shows all designated special purpose registers.

Of the interrupt-related registers, Register 2 is used to (un)mask critical sections of the RIPE program from external system-issued interrupts. For example, as shown in **pipe** (Fig. 4), the interrupts are only enabled after the task has suspended, while they are masked during normal operation. Register 3 can be programmed to hold the task ID of the next task to be loaded and run on the RIPE device out of the available task pool. Register 5 allows the RIPE program to assert "software interrupts." The RIPE model instantly reacts to unmasked hardware or software interrupts by loading the program and register set corresponding to the next task to be emulated, which is identified by Register 3. The usage of the special registers will be shown in Section III-B.

B. Programming Language and Assembler

To better understand the programming model of the RIPE device, Fig. 5 presents the main structure of a program to model the **IO** application introduced in Section II. Statements starting with a semicolon (;) are inlined comments.

The RIPE program starts with a header describing the core and the task identifier: `MASTER`[⟨`coreID`⟩, ⟨`taskID`⟩]. All of the tasks running on any given IP core are described within a single program, so that there is one program per RIPE device. A RIPE program can contain one or more task description. Recall that **IO** models an application with a linear program flow, which can be suspended by the OS to process I/O interrupts. Therefore, two tasks are described: task #0 (the main application) and task #1 (the interrupt handler) within the same master IP (core ID 1).

The next few statements express initialization of the register file for this task. Unique labels should be used for register names/tags. This allows correct initialization and easy identification of the registers within the program. For task #0, the main body of the RIPE program, in which the execution flow is linear, is composed of sequences of reads and writes interleaved with register accesses (mostly, to set up transaction addresses

```
MASTER[1, 0]                          ; Main application (Task 0)
 ; Special Registers
   REGISTER IntrpMaskReg 0            ; Unmask interrupts
   REGISTER TaskIDReg 1               ; Next task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0xd0abcdef        ; Initialize address GPR
   REGISTER DataReg 0                 ; Initialize data GPR
   ...
BEGIN                                 ; Comments
   ; Normal application flow
   Idle(10)                           ; Idle for 10 cycles
   Read(AddrReg)                      ;
   ...
   SetRegister(AddrReg, 0x10fedcab)   ; Setup an address
   SetRegister(AddrReg, 0x10abcdef)   ; Setup a data value
   Write(AddrReg, DataReg)            ;
   ...
END                                   ;
MASTER[1, 1]                          ; I/O driver task (Task 1)
 ; Special Registers
   REGISTER IntrpMaskReg 0            ; Unmask interrupts
   REGISTER SWIntrpReg 0              ; Disable SW interrupts
   REGISTER TaskIDReg 0               ; Next task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0                 ; Initialize address GPR
   REGISTER DataReg 0                 ; Initialize data GPR
   ...
BEGIN                                 ; Comments
 ; Interrupt Handling Routine
 IntrptHandler
  ; OS Suspension Routine
   SetRegister(IntrpMaskReg, 1)       ; Mask interrupts
   SetRegister(AddrReg, 0x30bebeef)   ; Setup an address
   Read(AddrReg)                      ;
   ...
  ; I/O Routine
   SetRegister(AddrReg, 0x30beefcd)   ;
   SetRegister(DataReg, 0x10101010)   ;
   Write(AddrReg, DataReg)            ;
   Idle(121)                          ;
   ...
  ; OS Release Routine
   ...
   SetRegister(SWIntrpReg, 1)         ; Trigger SW interrupt
   SetRegister(SWIntrpReg, 0)         ; Deassert SW interrupt
   Jump(IntrptHandler)                ; Get ready for next event
 ; End Interrupt Handling
END                                   ;
```

Fig. 5. RIPE program for the **IO** application.

and data). Flow-control instructions might be inserted where appropriate but are not needed for this application. Note the initialization of interrupt-related registers at the top of task #0; upon a hardware interrupt, the RIPE swaps the context to the task having the ID provided in `TaskIDReg`, i.e., to task #1 (the I/O interrupt handler). Since task #0 can be suspended by the OS to process I/O interrupts, `IntrpMaskReg` is set as unmasked, allowing for such suspension.

The OS-driven context-switch traffic and the I/O-handler routine are programmed in task #1. Within the interrupt routine (starting with label `IntrptHandler`), which is the critical section of the flow, interrupts are disabled (first instruction of the task body). At the end of the flow, a software interrupt is artificially triggered to restore the normal program flow to task #0. Upon another hardware interrupt in the main task, the interrupt-handler routine will be executed again from the top. The flow therefore mimics Fig. 3(d).

The RIPE program containing the aforementioned instructions must be transformed into a binary file for use within the RIPE device. An assembler tool takes care of this step, with a one-to-one correspondence between program instructions and binary opcodes. Within the binary, the individual task sections are appended in the order of their task ID. A header with a small-task lookup table is prepended.

During the setup phase, the RIPE device loads the binary and based on the information encoded at the start of the binary file, determines the number of tasks and the amount of program memory and the register file size to be allocated to each one.

IV. Generating RIPE Programs

In this section, we outline three ways of transforming application requirements into RIPE programs. One of the techniques will be discussed in detail to show feasibility of RIPE as a simulation aid and to create a validation environment for the RIPE-device accuracy.

A. Trace Parsing and Replay

In this scenario, as is shown in Fig. 1, the availability of a preexisting model for the IP under study is assumed. Here, the RIPE program generation goes through two steps. First, a reference simulation is performed by using the available IP models, and an execution trace for each IP master in the system is collected. The trace is a very straightforward log of events on the OCP pin out; entries include requests, responses, and interrupts, all of which are annotated with timestamps. A sample-trace snippet is presented in Fig. 7(a). Second, the trace is parsed with an offline tool. The output of the tool is the desired RIPE program. The resulting program is coded to behave exactly as the original IP model in the native system and to behave as the core would do when plugged to a different interconnect. This program is now ready to be used for cycle-accurate interconnect design-space exploration with extremely realistic test traffic.

This type of flow is useful whenever the preexisting IP model is not available, due to licensing or technical issues, for the next coexploration phase. In this case, the RIPE can provide a quick functional yet cycle-accurate port of the IP model to an MPSoC interconnect. Even if the IP model is available, a simulation speedup can be achieved without significant losses of accuracy (Section VI). The offline-parsing tool must, of course, have some knowledge about the traced application in order to correctly analyze and rearrange execution traces into RIPE programs. While this effort is not trivial, it is feasible and provides a path for validation of the presented RIPE device in a complete cycle-accurate flow, as described in Section V.

B. Trace Editing

In a related scenario, an IP model might be available, but it may differ with some respect from the IP that will eventually be deployed in the SoC device. In this scenario, the RIPE may be used to approximate the IP, as shown in Fig. 2. The approximation may be introduced by trace editing, for example,

by changing the number of bus transaction or the delay between transactions to model a cache subsystem. In this scenario, overall cycle accuracy with respect to the eventual system is, of course, not guaranteed. However, the RIPE will still be able to react with cycle accuracy to any optimization in the SoC interconnect. Provided that the transaction patterns are kept close to the ones of the target IP core, the approach will result in valuable guidelines.

C. Direct Development

Finally, RIPE programs can be written from scratch without reference IP traces. In this case, the flexible RIPE instruction set allows one for a full-featured traffic-generation system. The availability of built-in flow-control management lets the designer implement the same synchronization patterns, which are present in real-world applications (see Section III). Additionally, the application chunks enclosed within synchronization points can quickly be rendered by exploiting the flexible-loop structures provided by the RIPE ISA, thus providing capabilities at least on par with those of traditional stochastic traffic-generator implementations, as shown in [10], [16], and [19]. In the very first stages of development, the RIPE can also be deployed as a validation tool to check the correct functionality of the interconnect under the load of the supported transaction types. An alternate possibility, as demonstrated in [18], is using the RIPE as an interface between formal and simulation models in a hybrid environment. Here, the RIPE programs are written based on guidelines provided by the arrival curves obtained by formal-analysis methods. These programs are then used to generate communication events for the simulation environment. Thus, the versatility of our RIPE flow allows for deployment in a number of situations.

V. RIPE as a Simulation Aid

As an example of RIPE functionality, we now adopt the flow presented in Section IV-A to show its feasibility and to create a validation environment for the RIPE device accuracy.

A. Reference MPSoC System

For validation purposes, the RIPE model is integrated into the MPARM [22] reference system. MPARM is a homogeneous multiprocessor instruction-set simulation platform with a configurable number of processors as IP masters with private and shared memories and semaphore and interrupt devices. It also contains a port of Real-Time Executive for Multiprocessor Systems [3]—a real-time OS. The IP cores can be plugged onto one of several interconnect architectures, such as AMBA [7], STBus [29], and ×pipes [12]. The use of the OCP v2.0 protocol at the interfaces between the IP cores and the interconnect allows for easy exchange of native cores with RIPE blocks (Fig. 1). To record execution traces, the OCP interface modules within the MPARM system (the network interfaces in the case of ×pipes and the AMBA AHB bus master) were adapted to collect traces of OCP requests, responses, and interrupt events in a predefined file format (.trc).

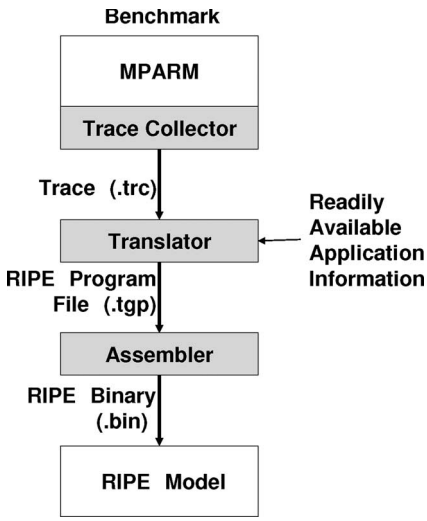


Fig. 6. Trace to RIPE program flow.

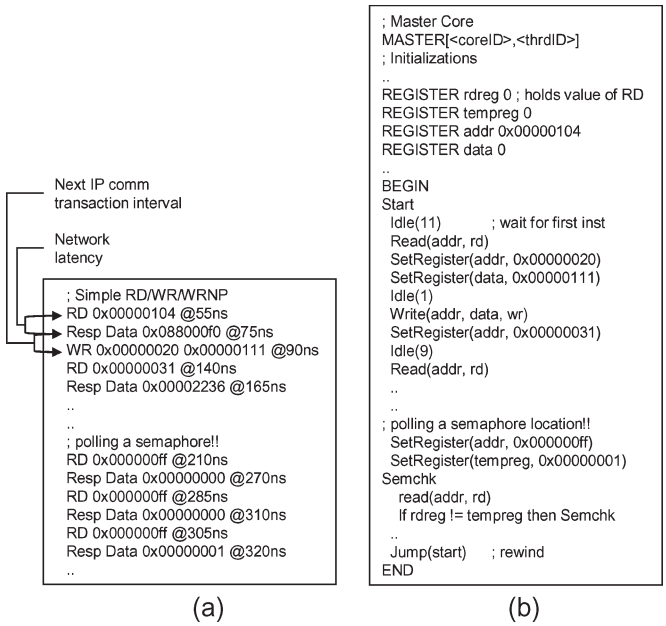


Fig. 7. (a) MPARM trace. (b) RIPE program.

It is worth stressing that modeling the communication patterns described in Section II is not trivial. The amount of annotations that can be extracted from the application and its traces reflects the programmer's degree of knowledge and access to the application-synchronization schemes, to the interrupt routines, and to the OS internals.

B. Trace to RIPE Program

The RIPE validation flow is illustrated in Fig. 6. During the reference simulation, traces are collected from all OCP interfaces in the system. The address and (if any) data fields of the transactions are also observed. Trace entries may contain one of many transaction types: single or burst read/write requests, assertion of hardware interrupt, arrival of response, etc. Fig. 7(a) shows an example trace.

The next step is to convert the traces into corresponding RIPE programs (.tgp). The offline translator tool outputs symbolic code; Fig. 7(b) shows the RIPE program derived from traces in Fig. 7(a). The automated algorithm in the conversion flow is capable of detecting and capturing many synchronization behaviors, without the need for the designer to handle them manually. We will explain the translator operation in Section V-C together with details regarding the required application insight, which is readily available to the designer. Finally, the assembler tool is used to convert the symbolic RIPE program into a binary image (.bin), which can be loaded into the RIPE instruction memory and executed. The entire flow is fully automated, and the time taken for this process is discussed in Section VI.

C. Automated Translation of IP Traces Into RIPE Programs

As discussed in Section II, some prior knowledge about the MPSoC system used in the reference simulation is required to accurately program the RIPE device. Apart from the sequence of transaction requests and responses, the following is a list of information needed for correct operation of the translator:

1) the global identifier of the IP core in the system;
2) the clock period of the IP core;
3) the addressing ranges representing semaphore (pollable) resources;
4) the timestamp of interrupt events;
5) the timestamp of the return from an interrupt-handling routine;
6) the timestamp of a spontaneous request for descheduling by an application.

The first three pieces of information are readily available design specifics and are encoded in the trace filename; the rest are explicitly or implicitly (provided some knowledge of the application functions) present within the trace file. For example, incoming interrupts are detected on the OCP pin out and explicitly recorded in the trace. On the other hand, returns from interrupt-handling routines must be located implicitly by detecting known behavior, such as a specific memory access at the end of the handler or at the return point in the main code.

We use the system traces given in Fig. 7(a) as an example source for transformation into a RIPE program and the result is in Fig. 7(b). Let the clock period be 5 ns and the semaphore location be `0x000000ff`. As shown in Fig. 7(b) and described in Section III-B, the RIPE program starts with the typical core identifiers. Register `RDReg` is defined as the name of the special register where the value of read transactions is stored (Table II).

At the beginning of the trace file, the first communication request, a read (`RD`), occurs at 55 ns, meaning that the RIPE has to wait 11 (55/5) cycles before issuing this transaction. Therefore, an `Idle` wait is observed in the RIPE program. When parsing this trace statement, the translator collects the `RD` address and initializes one of the registers marked as available in the register table (tagged as `addr` on top of the program). Based on the principle of "time-shifting" (Requirement #1) discussed in Section II-B, we ignore the response to this RD event at 75 ns but note the time interval of three [(90–75)/5] cycles to the next trace event, the `WR` at 90 ns. New values have to be set up in the address and data registers, which takes a cycle each (either for updating the already used `addr` and `data` or for setting up a new pair of registers). An ensuing `Idle` wait is added to fill the gap, then the `WR` instruction is appended.

This translation process continues until the trace entry at time 210 ns, when the semaphore address is encountered. By identifying the address as belonging to a semaphore location and knowing the polling behavior of the original IP core, the translator inserts the `Semchk` label and an `If` conditional statement. This statement checks whether the read value is equal to "1," which reflects an unlocked semaphore. This loop effectively models the semaphore polling behavior. The semaphore address and expected unlock value are set up prior to the loop label to avoid repeated initialization, thus allowing for continuous polling at the maximum frequency rate for unlimited periods. Idle waits can obviously be added in the loop should the original IP core have a low-frequency polling behavior. All master devices attempting to access this semaphore incorporate the same routine in their RIPE program, thus capturing the system dynamics to meet the Requirement #2.

Within the translator, a register-allocation algorithm correctly sets up all the required data in registers before the OCP or the flow-control instructions that need them are scheduled for execution. It is possible that streams of closely packed communication requests may leave few or no interleaved idle cycles available for preparing their address (and data, if any). In this case, the translation algorithm exploits the slack (idle-wait time) available further above in the transaction sequence for setting up register values ahead of time. However, in case of very long streams of back-to-back writes, a lack of free registers may occur. In this case, the size of the register file must be increased to avoid an accuracy loss due to hiccups in the sequence of writes. We expect the problem to occur with minimal frequency, as two idle cycles among transaction entries are enough to allow for streams of arbitrary length. The problem is of no importance in the context of a simulation RIPE device (as in this paper) but would have an area penalty in a hardware implementation.

D. Handling-Interrupt Reactiveness

For modeling interrupt routines and OS internals (Requirement #3), specific locations within the trace file, such as interrupt-handling-routine entry and exit points, have to be recognized by the translator tool to optimally insert the corresponding code as a task into the RIPE task pool. The trace files are always annotated with the time of occurrence of interrupt events. However, the exit points need to be carefully screened, since they depend on the degree of cooperation between the application and the OS.

Using the **pipe** example (Fig. 4), let us consider this aspect in more detail. Here, the task is explicitly interacting with the OS internals, as described in Section II. Usually, this interaction can be achieved by OS API calls, without direct access to the interrupt handler code, whose exit point is therefore assumed to be not accessible to the programmer. As a result, the only annotations of significance within the trace file are the synchronization points (semaphore checks) and the interrupt arrival time. The RIPE program thus mimics the flow shown in Fig. 4,

first by reading the semaphore location, then choosing to continue or suspend depending on the lock. Upon resumption by a hardware interrupt, a final (re-)check of the semaphore unlock is done to ensure safe task operation. In the RIPE program, this is realized via three tasks; the dotted lines in Fig. 4 mark their boundaries. The primary task represents the main application flow. The interrupts are masked here, as the application is insensitive to hardware interrupts unless in suspension state. If the semaphore is found locked, the flow is routed to load the OS routine, which leads the processor to an idle wait. The translator captures the chunk of trace after the semaphore check in an independent OS task, which always yields control to a third task consisting of an infinite loop of idle-wait instructions. The easily identifiable sequence of transactions between the eventual arrival of the hardware interrupt and the semaphore recheck is the OS wake-up routine to reschedule the suspended main program, and the translator appends it as a part of the OS task. In the RIPE program, hardware interrupts are used to wake up from the suspension state within OS routines, while software interrupts redirect the execution flow toward the main task. Note that `IntrpMaskReg` is set to "masked" for the regular program and OS execution and is only unmasked within the suspension task.

After performing the translation described in this section and after RIPE-program assembling, a second set of simulations can be run on a platform with RIPE and a variety of interconnects, thereby evaluating performance of interconnect-design alternatives.

VI. Validation Results

The outcome of the validation process should show that the requirements outlined in Section II are sufficient to extract IP traffic patterns in a manner which is accurate yet independent of interconnect characteristics. For this purpose, we simulate different applications within the MPARM framework, first using the native [Advanced RISC (Reduced Instruction Set Computer) Machines (ARM)] cores and then using the RIPE model, and compare the resulting benchmark statistics. We undertake this experiment for six benchmarks. Each is tested with 1–12 (1P–12P) system processors with cache (see Fig. 1) simultaneously plugged to the system interconnect. The aim is to ascertain the accuracy of the RIPE model, device, and translation framework when stressed by complex transaction patterns.

Five of the benchmarks are based on the examples introduced in Section II: **matrix**, **poll**, **multi**, **IO**, and **pipe**. One more application (**"cacheloop"**) is added as a reference to make the validation more comprehensive. The application **cacheloop** is a dummy program, which continuously performs cache fetches. As such, it is generating no bus transactions, except for a few at boot and shutdown. It is intended as a metric of the maximum simulation-time speedup achievable by the replacement of IP cores with another simulation device.

In the first experiment, we only aim at validating the trace collection and translation. Fig. 8 outlines the process. We run the same benchmarks over two of the interconnects of MPARM, namely, AMBA AHB [1] and the ×pipes [28] NoC. As expected, we measure very different execution times due to the

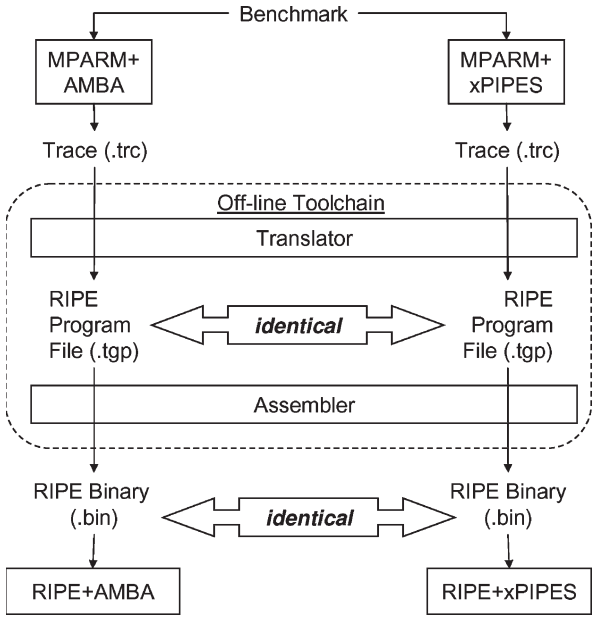


Fig. 8. RIPE accuracy-validation test.

different interconnect features, and the execution traces reflect these differences. However, after translation, a check across .tgp programs shows no difference at all, because the network latency factor is completely abstracted from in the RIPE programs. As a consequence, a trace collected on one interconnect is indeed usable to generate a program to be run on another. This result validates our approach and strengthens the postulate of the requirements outlined in Section II, which decouples simulation of the IP cores and of the underlying interconnect.

We now proceed to measuring the accuracy of the our design flow, i.e., how well the RIPE programs extracted from ARM execution traces match the original execution. Table III summarizes the results of simulations[1] done on the AMBA AHB interconnect with ARM processors from MPARM and then with RIPE devices. The columns report the overall execution time of the benchmarks (in clock cycles) and the number of single-read (SR), single-write (SW), and burst-read (BR) transactions observed on the bus. The column "Inaccuracy" is a measure of the relative difference in simulated cycles and bus accesses when replacing ARM cores with RIPE devices.

The table shows that replacing ARM processors with RIPE devices yields excellent precision, with inaccuracies close to 0% in most cases, resulting in a faithful reproduction of the original execution flow and traffic pattern. The inaccuracies in the SR count and the execution time in **poll** are due to the compounding of minimal timing mismatches caused by the semaphore polling mechanism in RIPE programs. In the real system, the first few semaphore polls are found to occur at a slightly different rate than subsequent ones, due to assembler-level and caching effects. Eventually, polling occurs at periodic intervals. This initial timing mismatch is not captured in the RIPE model, which performs all polling loops at the asymptotic

[1]Benchmarks taken on a Pentium 4 2.26 GHz with 1 GB of RAM. The absence of disk-swapping effects is checked during simulation. In particular, for benchmarks with a short duration, time measurements are taken by averaging over multiple runs, and care is put in minimizing disk-loading effects.

TABLE III
RIPE VERSUS ARM PERFORMANCE ON AMBA. THE TREND IS SIMILAR FOR 6, 10, AND 12 IP SYSTEMS

Benchmarks	# IPs	RIPE				MPARM				Inccuracy	
		Execution Cycles	SR	SW	BR	Execution Cycles	SR	SW	BR	Exec %	SR %
cacheloop	2	2500916	0	32	51	2500908	0	32	51	0.000%	0.000%
	4	2501721	0	64	106	2501714	0	64	106	0.000%	0.000%
	8	2503321	0	128	2018	2503314	0	128	201	0.000%	0.000%
matrix	2	1324711	0	117502	186	1324717	0	117502	186	0.000%	0.000%
	4	1326582	0	235004	374	1326588	0	235004	374	0.000%	0.000%
	8	1421281	0	470008	7502	1421272	0	470008	750	0.001%	0.000%
poll	2	881839	7176	71764	254	883977	7201	71764	254	0.242%	0.347%
	4	975267	18241	143596	508	976488	18183	143596	508	0.125%	0.319%
	8	1139110	46044	287356	1016	1140199	46300	287356	1016	0.096%	0.553%
multi	2	1823882	14	85729	24764	1824135	14	85729	24764	0.014%	0.000%
	4	2224333	42	192745	52242	2225867	42	192745	52242	0.069%	0.000%
	8	3482223	98	407707	109820	3482793	98	407707	109820	0.016%	0.000%
IO	2	1156047	2560	68494	18271	1158639	2560	68495	18271	0.224%	0.000%
	4	1446888	2560	145826	36966	1449109	2560	145827	36966	0.153%	0.000%
	8	2325228	2560	300514	74435	2325625	2560	300515	74435	0.017%	0.000%
pipe	2	745386	2601	56004	16293	754998	2601	56004	16293	1.273%	0.000%
	4	1051512	5246	114118	33257	1055056	5247	114298	33313	0.336%	0.019%
	8	1829005	10530	229675	66321	1833183	10530	229675	66321	0.228%	0.000%

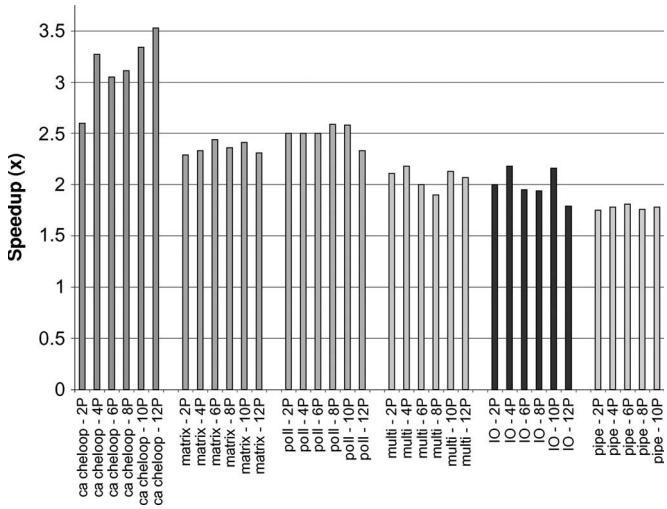


Fig. 9. RIPE versus MPARM speedup.

rate. This causes RIPE to be affected by a small timing skew, which impacts subsequent simulation.

The inaccuracies in interrupt-related benchmarks are due to minor issues in properly pinpointing different sections of OS code in the execution trace, as discussed before in Section V. The near-matching statistics, however, fully prove the role of the RIPE as a powerful design tool to mimic complex-application behavior in replacement of a real IP core.

Scalability tests, based on simulation time in seconds, performed by increasing the number of processors attached to the bus, exhibit two main different trends, as shown in Fig. 9. The **cacheloop** exhibits a fundamentally monotonic trend, showing the advantage of replacing a progressively increasing amount of system cores with a faster device model. Other benchmarks show a fundamentally constant figure, or an asymptotic increase (for example, **matrix**). This seemingly strange behavior can be explained as follows. An increase in the number of processors implies more traffic on the interconnect, thereby shifting the simulation load toward the interconnect model. At a certain point, the interconnect becomes completely saturated. In this condition, no further speedup is achievable because the simulation time of ARM processors is, anyway, mostly spent in idle waits for bus responses—leaving no room for improvement to RIPE devices, regardless of their efficiency. To support this analysis, we observe that the lowest speedup is achieved for **pipe**, which is also found to be the benchmark with the highest bandwidth requirements, and therefore, the highest load on the interconnect model. We would like to stress that, as the **cacheloop** demonstrates, this decrease in simulation speedup is not a shortcoming of our RIPE approach and is, instead, a direct consequence of benchmark and system behavior. In absolute terms, a gain of 1.75× to 3.53× is observed when running the benchmark code on RIPE devices as opposed to ARM processors, owing to the removal of the computation logic within cores. It is noteworthy that, even though speedup is not the primary objective of RIPE, it compares favorably to previous work in the area (a speedup of 1.55× is reported in [25]), particularly given the fact that it is achieved at the cycle-true level of abstraction.

The time penalty for trace collection is small and is incurred only once. For example, when running the relatively complex **pipe** benchmark on the AMBA interconnect with four ARM processors, a benchmark run augmented to collect reference traces takes 20 s, and subsequent translation and elaboration requires an additional 12 s for a 5.6-MB trace file. Only one such iteration is needed to validate the RIPE model and for subsequent design-space exploration. Additionally, since processed RIPE programs are identical regardless of the reference interconnect in which raw traces are collected, such a collection could be performed on top of a fast transactional interconnect model, further reducing the impact of the reference simulation.

VII. RIPE AS A DESIGN TOOL

To demonstrate the potential of RIPE as a coexploration tool, we look at a variant of the **multi** application, first discussed in Section II-C, in more detail. Specifically, we consider a

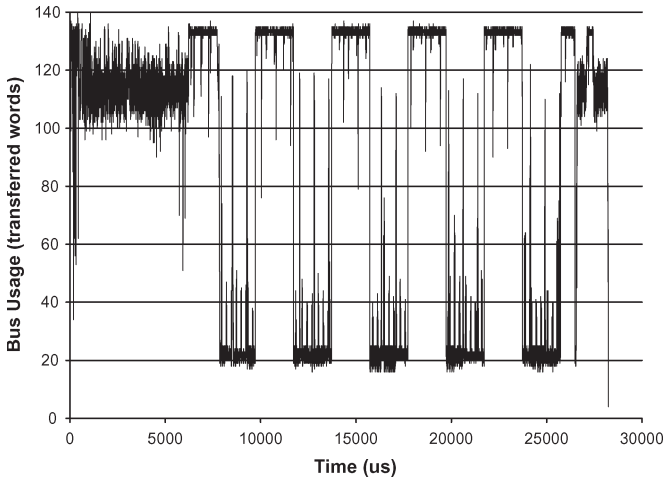


Fig. 10. Reference traffic pattern.

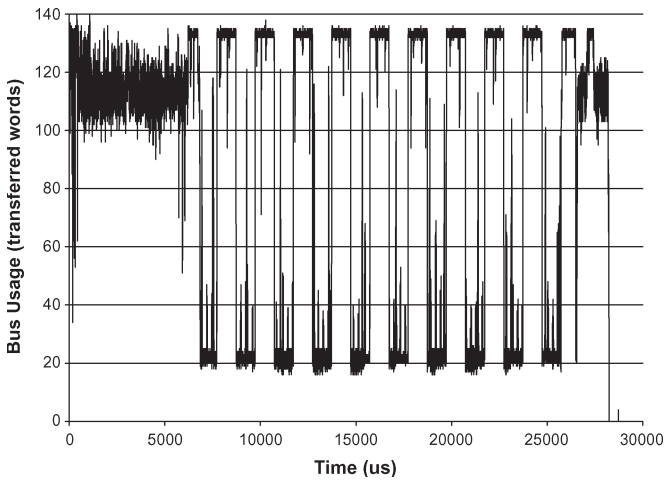


Fig. 11. Case I.

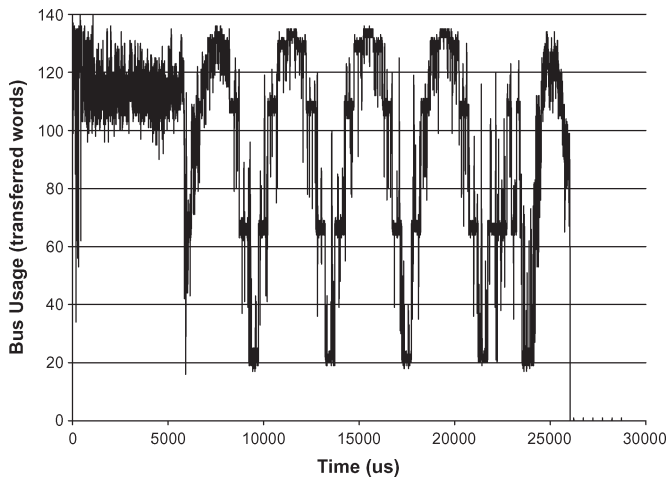


Fig. 12. Case II.

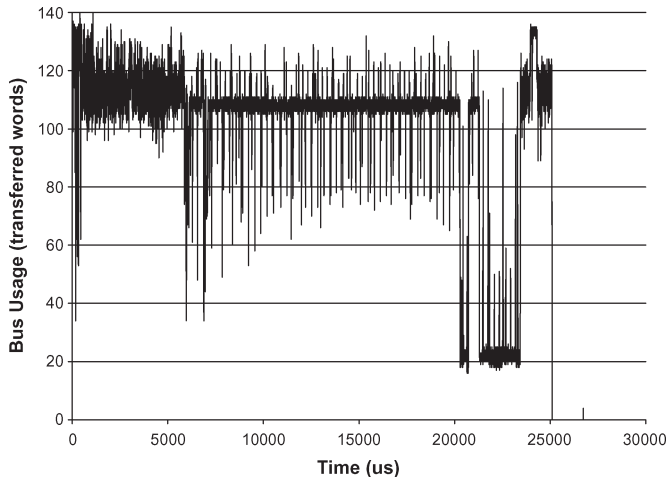


Fig. 13. Case III.

five-processor bus-based system with one RIPE configured to act like a timer device. This core triggers the delivery of interrupts at regular intervals to the other four RIPE devices, which, as a result, switch among two tasks. The two tasks are tuned to have very different bandwidth requirements: One task performs matrix manipulations (**MM**) and heavily relies on data caches to minimize memory transactions, whereas the second task performs streams of writes (**WS**) to a memory attached to the bus. The **WS** task is very demanding on the interconnect and can easily saturate it, therefore impacting overall system performance. In MPARM, interrupts are triggered by writing to a specific address of a memory-mapped device; therefore, to trigger the interrupts that should come from a timer device, we write a small RIPE program issuing OCP writes at the right times. In turn, this is achieved by parameterized idle waits. Such a program is written in a dozen lines of RIPE code.

In this case study, using the RIPE, we test the behavior of this system for different interrupt delivery policies and study the resulting traffic profiles (Figs. 10–13). This type of exploration may be useful to schedule bus accesses for real-time tasks in critical systems. The traffic plots show the profile of the bus traffic over time, expressed as transferred data words over a time window of 2 μs.

In all the plots, until about the 6000-μs mark, the bus activity during the OS boot is observed. The boot activity is irregular but, on average, quite intensive in terms of required bandwidth, since all the processors are loading the OS and application instructions from the memory across the interconnect. After this mark, application code begins to be executed. In Fig. 10, a straightforward scheduling policy is used: A timer interrupt is sent to each core simultaneously, therefore causing all of the cores to switch among **MM** and **WS** at the same time. Since interrupts simultaneously reach all processors, all of them are in the same task group during any given time slice of execution. As expected, the bus load shifts depending on the task characteristics; the traffic profile exhibits a clear alternating pattern among two disproportionate usage values, with peaks above 130 and a floor of around 20 transactions per time window. The number of transitions between these two limits and the width of each peak correspond to the number of issued interrupt events and the interval between them (see Table IV). The tail of the plot is representing shutdown code and is not relevant.

Since excessive contention inflates the response latency of the bus, therefore decreasing performance, the traffic profile must be reshaped to decrease congestion. As is observed in

TABLE IV
INTERRUPT-ISSUE FREQUENCY FOR FOUR DIFFERENT MULTITASKING PATTERNS

	Interval among interrupts to same core (ms)	Notes
Reference	2	
Case I	1	
Case II	2	Processors receive interrupts staggered by a 0.5 ms offset
Case III	2	Two processors receive an extra interrupt just after the boot

Fig. 11, as compared to Fig. 10, doubling the interrupt-issue frequency does little to mitigate the bus-congestion issue; it only shifts the contention to a different time slot. Execution time remains constant at about 28 200 μs.

Let us now consider the impact on the bus activity of staggering the interrupt events. In Fig. 12, an interrupt is sent every 500 μs, but two interrupts to the same processor are spaced 2000 μs apart. The traffic profile is smoother; owing to staggering, **MM** tasks on some cores run in parallel to **WS** tasks on other cores. Over time, the system shifts from running four **MM** tasks to running four **WS** tasks and back, which results in a sinusoidal-like trend with visible steps. Peak congestion is only reached during a shorter fraction of the time, therefore reducing the execution time to about 26 000 μs.

To balance the traffic even better, the clear choice is to always overlap two **MM** and two **WS** tasks. This is achieved in Fig. 13, where two processors are forced to perform a context switch just after the OS boot, and the subsequent interrupt pattern is the same as in Fig. 10. Owing to much better traffic balancing, the bus never saturates, providing good performance and decreasing the execution time to 25 200 μs.

In Fig. 14, the benchmark execution time and the average communication latency for a write transaction on the bus are plotted for the four configurations. As shown, Case I exhibits basically identical performance to the reference, while Case II improves 18% on communication latency (and, thus, 8% on execution time) and Case III improves 24% on latency (and, thus, 11% on execution time). Therefore, Case III is the best among the alternatives under evaluation.

These experiments highlight that RIPE can be an extremely useful tool to explore communication bottlenecks even without having the real IP cores and benchmarks attached to the interconnect. The flexibility guaranteed by the interrupt handling support provides the designer with additional degrees of freedom and accuracy, allowing a realistic system exploration even in the presence of complex communication and synchronization patterns.

VIII. RELATED WORK

The use of IP emulation devices, such as traffic generators (TGs), is not new and several approaches and models have been proposed.

In [19], a stochastic TG model is used for the interconnect exploration; the IP behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. A similar approach in [32] uses random and semideterministic

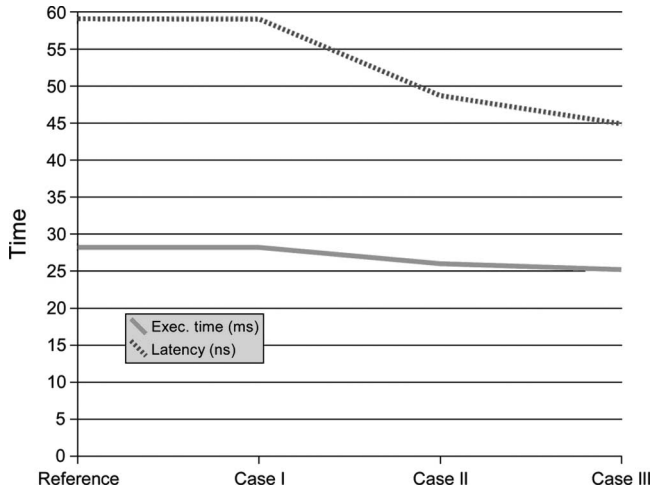


Fig. 14. Performance of the four synchronization patterns under test.

distributions. The IP model used for NoC optimization in [10] takes into account the nature of MPSoC traffic, such as real-time, short-data access, bursty, etc.; however, the injection rate is governed by statistical methods. In [31], an extra dimension of "self-similarity" is added to the stochastic model, which is argued to assist in precise characterization of multimedia traffic by examining the correlations in traffic traces at the macroblock level. Despite the refinements, the inherent probabilistic nature of the statistical approaches makes it less accurate, as each TG injects traffic in complete isolation from every other. As surveyed in [8], such stochastic models are widely popular for analysis of macronetworks, e.g., the Internet, that exhibit such behavior, which is, unfortunately, unlikely in an MPSoC environment.

A modeling technique which adds functional accuracy and causality is transaction-level modeling (TLM), which has been widely used for MPSoC design [11], [14], [17], [24], [25], [27]. In [24] and [25], TLM has been used for bus-architecture exploration. The communication is modeled as read and write transactions toward the bus. Depending on the required accuracy of the simulation results, timing information, such as bus arbitration delay, is annotated within the bus model. In [25], an additional layer called "Cycle Count Accurate at Transaction Boundary" is presented. Here, the transactions are issued at the same cycle as that observed in bus-cycle-accurate models. Intratransaction visibility is traded off for a simulation-speed gain. An average speedup of 1.55$\times$ is reported. While modeling the entire system at TLM, both [24] and [25] present a methodology for preserving accuracy with gain in simulation speed. Such models are efficient in capturing regular communication behavior, but the fundamental problem of capturing system unpredictability in the presence of OS and interrupts is not addressed.

In [22] (MPARM) and [15], complete cycle-true MPSoC systems including the full instruction set of the IP cores and the OS are described. This consequently impacts the simulation speed and the scalability of the system. Furthermore, the time required to investigate the performance impact of relatively minor changes in systems modeled in such a way often becomes major due to long implementation and simulation

times. However, as seen before, it presents an ideal tool for validation of our RIPE approach. To overcome the speedup limitation of such simulation-based approaches, an FPGA-based emulation platform has been proposed in [16]. However, the approach uses the stochastic or the trace-driven model to generate traffic, which, as addressed before, is not sufficiently accurate for MPSoC performance optimization. A transformation methodology of high-level simulation traces with cycle-true information from the target architecture, e.g., memory distribution and communication details, is presented in [20] and [21]. In [21], based on accurate information, different rules are specified for inserting and ordering synchronization events in the output-execution trace, while, in [20], a trace-based communication graph is adjusted with interconnect-specific details, such as connection-setup time, burst size, etc. The RIPE approach, in contrast, is to identify synchronization events based on system information and abstract it for communication refinement. As has been demonstrated, this has been successfully achieved for complex benchmarks with interrupts and OS. We believe that the RIPE model and the approach presented in [21] are complementary in addressing trace-based MPSoC analysis from functional to cycle-true abstraction.

In [26], a commercial TLM-based reactive-workload-generation framework is presented that is somewhat similar to our RIPE approach, wherein users can configure traffic patterns for handling synchronization and inter-IP events. Primitives for timing-dependent behavior are provided, so that the user can trigger actions which do not depend on application flows but on simulation time. The RIPE approach, however, supports multithreading, which is required for interrupt-driven OS context switches, and traffic generation at multiple levels of abstraction, including in a cycle- and bit-true environment.

Other commercial efforts also exist, including the OpenVERA [30] language and tool chain, that model concurrence and synchronization. However, our approach is focused on maximum accuracy of results, while OpenVERA is mostly focused on the verification issue, providing a flow from higher abstraction levels to RTL.

IX. Conclusion

In this paper, we identified the requirements to split the design of computation and communication entities in an MPSoC design. Modeling requirements derive from real-life applications, and they represent complex execution scenarios, including an OS layer and asynchronous interrupt-based synchronization. The key piece of the puzzle can be identified in reactiveness to external events and system state. The RIPE device presented here and its programming interface provide support for the identified scenarios of traffic modeling and generation. Unlike traditional or commercial approaches, extensive support for multithreading is provided, and via flow-control instructions, it is possible to model a large and representative class of traffic types observed in MPSoCs. We have shown the usefulness of the RIPE device within different coexploration domains, either to replace existing IP cores in new domains or to provide emulation of IP cores that are under development or, even yet, to be designed. Experimental results show excellent accuracy figures when validating the RIPE against a reference system and a respectable gain in simulation speed when taking into account previous literature and the cycle-accurate abstraction level. A case study is supplied to show the usefulness of RIPE in a design-space-exploration context. Future work may include carrying the current RIPE design toward silicon for on-chip traffic generation and toward behavioral models for extra simulation speedups.

Acknowledgment

The authors would like to thank the anonymous reviewers for extensive comments that have helped in vastly improving this paper.

References

[1] The Advanced Microcontroller Bus Architecture (AMBA) homepage. [Online]. Available: www.arm.com/products/solutions/AMBAHomePage.html

[2] "The SystemC discussion forum," *Web Forum*. [Online]. Available: www.systemc.org

[3] *The Real-Time Operating System for Multiprocessor Systems*. [Online]. Available: http://www.rtems.com

[4] *Open Core Protocol Specification*, 2003. Release 2.0.

[5] F. Angiolini, S. Mahadevan, J. Madsen, L. Benini, and J. Sparsø, "Realistically rendering SoC traffic patterns with interrupt awareness," in *Proc. IFIP Int. Conf. VLSI-SoC*, Sep. 2005.

[6] ARM, *AMBA AXI Protocol Specification*. (2004, Mar.). version 1.0. [Online]. Available: www.arm.com

[7] ARM Holdings PLC, *Advanced Microcontroller Bus Architecture (AMBA) Specification rev 2.0*, 2001.

[8] S. Avallone, A. Pescape, and G. Ventre, "Analysis and experimentation of Internet traffic generator," in *Proc. FTDCS*, 2004.

[9] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[10] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," in *J. Syst. Archit.*, vol. 50, Feb. 2004, pp. 105–128.

[11] L. Cai and D. Gajski, "Transaction level modeling in system level design," Center for Embedded Comput. Syst., Inf. Comput. Sci., Univ. California, Irvine, CECS Tech. Rep. 03-10, Mar. 2003.

[12] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "×pipes: A latency insensitive parameterized network-on-chip architecture for multi-processor SoCs," in *Proc. ICCD*, 2003, pp. 536–539.

[13] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. 38th Des. Autom. Conf.*, Jun. 2001, pp. 684–689.

[14] F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato, "A timing-accurate modeling and simulation environment for networked embedded systems," in *Proc. 42th DAC*, Jun. 2003, pp. 42–47.

[15] F. Fummi, S. Martini, G. Perbellini, M. Poncino, F. Ricciato, and M. Turolla, "Heterogeneous co-simulation of networked embedded systems," in *Proc. DATE*, Feb. 2004, pp. 168–173.

[16] N. Genko, D. Atienza, G. D. Micheli, L. Benini, J. M. Mendias, R. Hermida, and F. Catthoor, "A novel approach for network on chip emulation," in *Proc. Int. Symp. Circuits Syst.*, 2005, pp. 2365–2368.

[17] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[18] S. Kuenzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *Proc. DATE*, Mar. 2006, pp. 236–242.

[19] K. Lahiri, A. Raghunathan, and S. Dey, "Evaluation of the traffic-performance characteristics of system-on-chip communication architectures," in *Proc. 14th Int. Conf. VLSI Des.*, 2001, pp. 29–35.

[20] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 6, pp. 768–783, Jun. 2001.

[21] P. Lieverse, P. van der Wolf, and E. Deprettere, "A trace transformation technique for communication refinement," in *Proc. 9th Int. Symp. Hardware/Software Codesign (CODES)*, Apr. 2001, pp. 134–139.

[22] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proc. DATE*, 2004, pp. 752–757.
[23] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparsø, and J. Madsen, "A network traffic generator model for fast network-on-chip simulation," in *Proc. DATE*, Mar. 2005, pp. 780–785.
[24] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai, "A practical approach for bus architecture optimization at transaction level," in *Proc. DATE*, Mar. 2003, pp. 176–181.
[25] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proc. 38th DAC*, 2004, pp. 113–118.
[26] S. Schneider, U. Mueller, and D. Tiegelbekkers, "A reactive workload generation framework for simulation-based performance engineering of system interconnects," in *Proc. MASCOTS*, Sep. 2005, pp. 484–487.
[27] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-chip interconnect woes through communication-based design," in *Proc. 38th DAC*, Jun. 2001, pp. 667–672.
[28] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. D. Micheli, "×pipes lite: A synthesis oriented design library for networks on chips," in *Proc. DATE*, Mar. 2005, pp. 1188–1193.
[29] STMicroelectronics, *The ST Bus*, 2004. [Online]. Available: http://www.st.com/stonline/
[30] Synopsys, *OpenVERA Technology Backgrounder*, 2001. White paper. [Online]. Available: http://www.open-vera.com/
[31] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 1, pp. 108–119, Jan. 2004.
[32] D. Wiklund, S. Sathe, and D. Liu, "Network on chip simulations for benchmarking," in *Proc. 4th IEEE Int. Workshop Syst.-on-Chip for Real-Time Appl. (IWSOC)*, 2004, pp. 269–274.
[33] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. San Mateo, CA: Morgan Kaufmann, 2001, ch. 3.

Shankar Mahadevan (M'05) received the M.S. degree in electrical engineering from Virginia Polytechnic Institute and State University, Blacksburg, in 2002 and the Ph.D. degree from the Department of Computer Science and Engineering, Technical University of Denmark, Lyngby, Denmark, in 2006.

His research interests are in the area of modeling embedded systems and interconnects at multiple levels of abstraction.

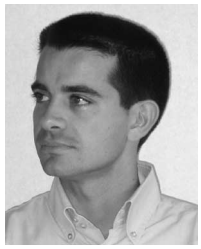
Federico Angiolini (S'07) received the M.S. degree (*summa cum laude*) in electrical engineering from the University of Bologna, Bologna, Italy, in 2003, where he is currently working toward the Ph.D. degree in the Department of Electronics and Computer Science.

His research is mostly focused upon memory hierarchies, multiprocessor embedded systems, networks-on-chip, and nanotechnologies.

Jens Sparsø (M'98) received the M.Sc. degree from the Technical University of Denmark (DTU), Lyngby, Denmark, in 1981.

Since 1982, he has been with the Section for Computer Science and Engineering, Department of Informatics and Mathematical Modelling, DTU, first, as an Assistant Professor and then was appointed as an Associate Professor in 1986 and Professor in 2007. His research interests are in architecture and design of very large scale integration systems, application-specific computing structures and processors, low-power design techniques, design of asynchronous circuits and systems, and communication structures for systems-on-chip (i.e., networks-on-chip).

Prof. Sparsø has been on the steering committees and technical program committees for several conferences. He was the General Chair for PATMOS 1998 and the Program Chair for PATMOS 1999 and ASYNC 2006. He was the Director and Local Organizer of a summer school on asynchronous circuit design at DTU in 1997. He is the coauthor of the book *Principles of Asynchronous Circuit Design-A Systems Perspective* (Kluwer, 2001). He was the recipient of the Radio-Parts Award and the Reinholdt W. Jorck Award in 1992 and 2003, in recognition of his research on integrated circuits and systems. He was the recipient of the Best Paper Award at the IEEE International Symposium on Asynchronous Circuits and Systems in 2005.



Luca Benini (S'94–M'97–SM'04–F'06) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1997.

He is currently a Full Professor with the Department of Electrical Engineering and Computer Science, University of Bologna, Bologna, Italy. He also holds a visiting faculty position with Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland. His research interests are in the design of system-on-chip platforms for embedded applications. He is also active in the area of energy-efficient smart sensors and sensor networks, including biosensors and related data-mining challenges. He has published more than 350 papers in peer-reviewed international journals and conferences, four books, and several book chapters.

Dr. Benini has been the Program Chair and Vice Chair of the Design Automation and Test in Europe Conference. He has been a member of the technical program committee and organizing committee of several technical conferences, including the Design Automation Conference, International Symposium on Low Power Design, and the Symposium on Hardware–Software Codesign. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF CIRCUITS AND SYSTEMS and the Association for Computing Machinery *Journal on Emerging Technologies in Computing Systems*.

Jan Madsen (S'83–M'90) received the M.Sc. degree in electrical engineering and the Ph.D. degree in computer science from the Technical University of Denmark (DTU), Lyngby, Denmark, in 1986 and 1992, respectively.

He has been with the Department of Informatics and Mathematical Modelling, DTU, as an Assistant Professor from 1992 to 1996, an Associate Professor from 1996 to 2002, and a Full Professor of computer-based systems since 2002. He has published 7 book chapters and more than 80 papers in peer-reviewed international journals and conference proceedings. His research interests are in modeling, analysis and design of embedded systems, particularly system-level tools for performance analysis and verification, hardware/software codesign, and wireless sensor networks.

Prof. Madsen has been the Program Chair and Vice Chair of the Design, Automation, and Test in Europe Conference and the Program Chair and General Chair of the Hardware/Software Codesign Conference. He has been a member of the Technical Program Committee and Organizing Committee of several technical conferences, including the Symposium on Hardware/Software Codesign, the International Symposium on System Synthesis, the Design Automation Conference, the Real-Time System Symposium, and the International Symposium on Industrial Embedded Systems.