A Traffic Injection Methodology with Support for System-Level Synchronization

Shankar Mahadevan[1], Federico Angiolini[2],
Jens Sparsø[1], Luca Benini[2], and Jan Madsen[1]

[1] Informatics and Mathematical Modelling,
Technical University of Denmark,
Richard Petersens Plads,
2800 Lyngby, Denmark
{sm, jsp, jan}@imm.dtu.dk
[2] Dipartimento di Eletronica, Informatica e Sistemistica,
University of Bologna,
Viale Risorgimento, 2
40136 Bologna, Italy
{fangiolini, lbenini}@deis.unibo.it

Abstract. In highly parallel Multi-Processor System-on-Chip (MPSoC) design stages, interconnect performance is a key optimization target. To effectively achieve this objective, true-to-life IP core traffic must be injected and analyzed. However, the parallel development of MPSoC components may cause IP core models to be still unavailable when tuning communication performance. Traditionally, synthetic traffic generators have been used to overcome such an issue. However, target applications increasingly present non-trivial execution flows and synchronization patterns, especially in presence of underlying operating systems and when exploiting interrupt facilities. This property makes it very difficult to generate realistic test traffic. This paper presents a selection of application flows, representative of a wide class of applications with complex interrupt-based synchronization; a reference methodology to split such applications in execution subflows and to adjust the overall execution stream based upon hardware events; a reactive simulation device capable of correctly replicating such software behaviours in the MPSoC design phase. Additionally, we validate the proposed concept by showing cycle-accurate reproduction of a previously traced application flow.

1 Introduction

The design space exploration for the interconnect fabric is an important but time-consuming step in designing a multiprocessor SoC (MPSoC). Depending on the application and the processing cores, the communication architecture may need to support wide ranges of traffic patterns, from bandwidth-intensive transactions such as cache refills to latency-critical transactions such as semaphore accesses or interrupt events. Unfortunately, a reliable analysis and optimization process requires cycle-true IP simulation models of both cores and interconnects to be simultaneously available and ready to interoperate, which is only possible late in the design flow.

Mahadevan, S., Angiolini, F., Sparsø, J., Benini, L., Madsen, J., 2007, in IFIP International Federation for Information Processing, Volume 240, VLSI-SoC: From Systems to Silicon, eds. Reis, R., Osseiran, A., Pfleiderer, H-J., (Boston: Springer), pp. 145–161.

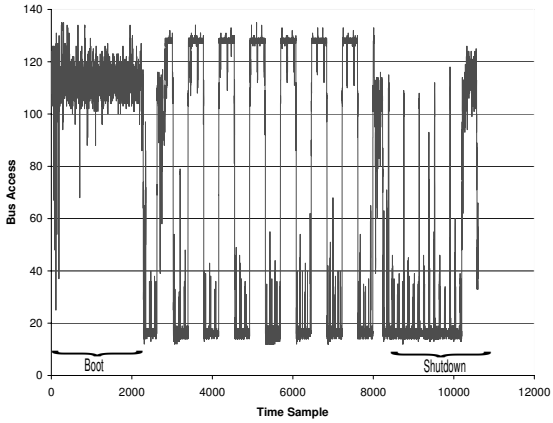

Fig. 1. Bus congestion over time for a multitasked application.

To cut on development time, Traffic Generators (TGs) are usually deployed instead of IP core models until the very last design stages. TGs can operate in a variety of ways, for example by creating synthetic traffic patterns according to some parameters (*e.g.* bandwidth and latency distributions), or by playback of prerecorded transaction traces collected on a reference system. Unfortunately, the former approach is only a gross estimation of the real traffic patterns that will be injected into the SoC, and fails to correctly capture the time distribution of traffic spikes which would occur in a real application. As for the latter approach, any prerecorded trace can be significantly different from the traffic that should actually take place, due to the eventual deployment of different cores and interconnect architectures. For example, synchronization by semaphore polling can require an unknown amount of bus accesses before getting lock ownership, and the resulting bus congestion is hard to model with traditional trace-based mechanisms. Our approach is significantly different; in that, we abstract away the computation aspect of the IP core, but realistically render externally observable communication behaviour, including responses to interrupt events.

Modelling application flows in response to inherently asynchronous communication events such as interrupts can be challenging, particularly, on a general-purpose processor, where it may involve Operating System (OS) interactions. While interrupts themselves typically have a low impact on communication resources, interrupt handling can cause severe network traffic peaks. For example, see Figure 1, where the bus usage over time is reported for a shared bus MPSoC. In the plot, in between a boot and a shutdown stage, it is easy to recognize a time-sliced multitasked benchmark where two tasks alternate; one of them has heavy bandwidth requirements, while the other one mostly operates in cache. Here, the context switch is triggered by an interrupt event, which subsequently causes a skew in the application flow. As this example shows, proper modeling of system tasks, including their communication and synchronization properties, is a key enabling factor in understanding their impact on interconnect resources, and consequently perform interconnect and system optimization. Any model describing IP core traffic should feature extensive *reactive capabilities*, to mimic the behaviour of the core even when facing unpredictable environmental events and net-

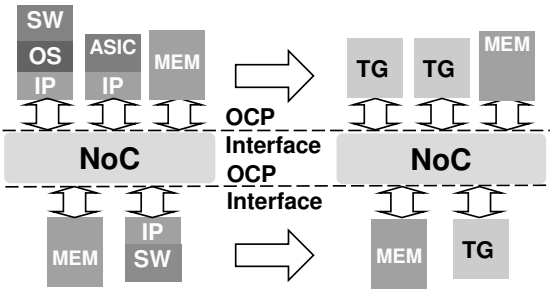

Fig. 2. Simulation Environment with bit- and cycle-true: (a) IP-cores, (b) TG model.

work performance, *e.g.* due to resource contention, bus arbitration and routing policies. Sample applications needing such complex modeling will be shown in Section 3.

In this paper, we present a *traffic generation model*, encompassing an instruction set and a programmable simulation device, that attempts to generate SoC traffic compliant with the behaviour of the IP cores that it is replacing (Figure 2). The proposed cycle-true TG approach allows the separation of computation and communication concerns, so that the designers can focus on accurate exploration of the SoC interconnect. This model allows both for the generation of synthetic traffic and for the reproduction of prerecorded traffic streams, but in any case is capable of realistically adjusting its output depending on complex external synchronization events, like semaphore interaction and interrupt notification. The TG device is a very simple instruction set processor, and is attached via a bit- and cycle-true OCP 2.0 [2] port to the SoC interconnect. Our approach is significantly different from a purely behavioural encapsulation of application code into a simulation device, in analogy with TLM modeling; we aim at faithfully replicating traffic patterns generated by a *processor running an application*, not just by the application. This includes *e.g.* accurate modeling of *cache refills*.

While the TG that we propose can be used in the same way as traditional TGs, a novel feature of our approach is that any knowledge about the behaviour of the actual system can be thoroughly taken into account and rendered by means of *TG programs*. The device programmability allows for the implementation of entire communication-dominated SoC applications on top of it, including ones that make use of OS facilities. Resulting traffic patterns closely resemble those of the real application running on top of the real IP core, while accurately handling the synchronization and intercommunication issues typical of multiprocessor systems. We focus both on the dynamics of core-initiated communication (reads, writes) [10] and on system-initiated messages, such as interrupts [3].

As a demonstration of the flexibility and accuracy of the model, we will show how the proposed flow can be applied to a complex test case, with general-purpose ARM processors running an OS in a multicore environment. The TG model is integrated into MPARM [9], a homogeneous multiprocessor SoC simulation platform, which provides a bit- and cycle-true SoC simulation environment and on which a port of the RTEMS [1] real-time OS is available. After performing a reference simulation, where execution traces were collected, we will process them to derive suitable TG programs capable of capturing fundamental application flow properties and synchronization

patterns. It is essential to notice that such programs are not passive translations of the original traces, but instead that they feature significant reactiveness to external events. By subsequently replacing ARM cores with traffic generators running such programs, we will analyze the accuracy of the proposed TG concept.

The rest of the paper is organized as follows. Section 2 presents related work. Relevant interrupt-aware applications to be modeled are discussed in Section 3. Section 4 presents details of the proposed implementation of the traffic generators, specifically stressing flow control handling in presence of interrupts. Section 5 describes possible ways to write programs for execution on top of TGs, and Section 6 highlights an example TG deployment flow. Section 7 presents simulation results which document the potential of our TG approach. Finally, Section 8 provides conclusions.

2 Previous Work

The use of traffic generators to explore NoC architectures is not new.

In [8], a stochastic model is used for NoC exploration. Traffic behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. Statistical approaches lack accuracy and can potentially exhibit correlations among system activities which are unlikely in a SoC environment. Further, asynchronous events such as interrupts are not easy to represent by these stochastic models. The simplicity and simulation speed of stochastic models may make them valuable during preliminary stages of NoC development, but, since the characteristics (functionality and timing) of the IP core are not captured, such models are unreliable for optimizing NoC features.

A modeling technique which adds functional accuracy and causality is transaction-level modeling (TLM), which has been widely used for NoC and SoC design [4, 5, 6, 11, 12, 14]. In [11, 12], TLM has been used for bus architecture exploration. The communication is modeled as read and write transactions which are implemented within the bus model. Depending on the required accuracy of the simulation results, timing information such as bus arbitration delay is annotated within the bus model. In [12] an additional layer called "Cycle Count Accurate at Transaction Boundary" (CCATB) is presented. Here, the transactions are issued at the same cycle as that observed in Bus Cycle Accurate (BCA) models. Intra-transaction visibility is here traded off for a simulation speed gain. While modeling the entire system at a higher abstraction level *i.e.* TLM, both [11] and [12] present a methodology for preserving accuracy with gain in simulation speed. Such models are efficient in capturing regular communication behaviour, but the fundamental problem of capturing system unpredictability in the presence of interrupts is not addressed.

In this chapter, we illustrate an accurate framework which is capable not only of modeling processor-initiated communication in presence of latency uncertainties [10], but even the processor behaviour when responding to fully asynchronous system events, such as interrupts. As is demonstrated in [13], the impact of interrupts can be significantly different for different OSs and network organizations. By providing cycle- and bit-true ports to the SoC communication backbone, and a few flow control instructions, we are able to accurately model the IP's reactiveness, which is essential

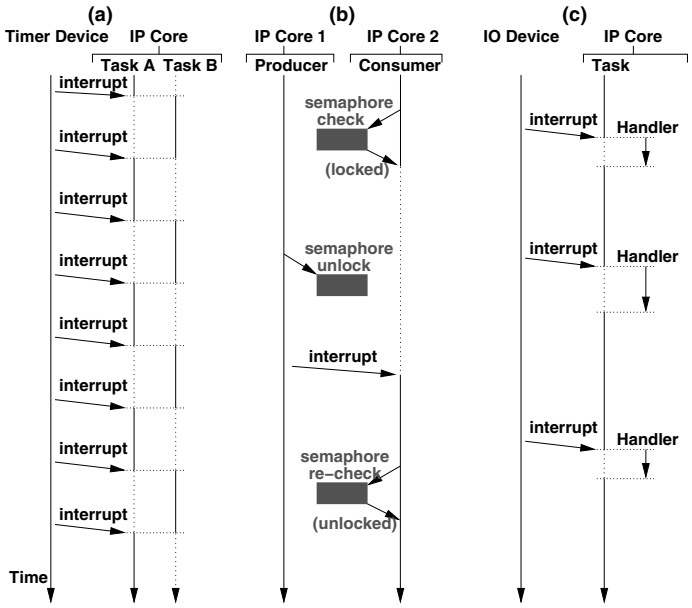

Fig. 3. Execution flow of interrupt-aware applications. Dotted lines represent suspended tasks.

for realistic fabric performance evaluation. Our methodology for application modeling, originally presented in [3], takes into account multitasking and the impact of an underlying OS, and is capable of representing a wide range of synchronization patterns. Additionally, we have deployed the flow in a test environment, and in Section 7 we will show this flow to be over 98% accurate and providing a speedup that, while nominal, favourably compares to [12].

3 Interrupt-Aware Synchronization Scenarios

Many communication and synchronization patterns are possible among tasks in a multiprocessor environment. This is especially true when interrupts are involved, since interrupts represent intrinsically asynchronous, system-initiated communication towards IP cores. To analyze such a wide variety of patterns [15], we identified three parallel applications, interacting both among their tasks and with the underlying OS, which highlight interrupt handling scenarios typical of real systems. These applications perform relatively light computation but exhibit non-trivial flow patterns, which makes them much more difficult to model than computation-intensive tasks. As such, these test cases are used to derive requirements of the most typical interrupt-based flow controls.

The application templates we identified are:

– A multi-tasking application (**"task"**), as in Figure 3(a). In this case, two tasks run on each processor; a variable amount of system processors may be present. No explicit communication is performed between tasks, neither intra- nor inter-core. The

context switching between tasks is performed by the OS in response to an external interrupt, which may typically be sent by a timer device. It is important to notice that, if tasks are asymmetric, any rescheduling translates into different traffic workloads for the communication fabric. This effect must be captured.

– A pipelined parallel application (**"pipe"**), as in Figure 3(b). For this case, a single task is mapped onto every system 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 to assess such availability. For example, the consumer checks a semaphore before accessing producer output. If this semaphore is found initially locked, a continuous polling might be attempted, but at the expense of wasted energy and saturation of the system interconnect. Instead, we implemented a mechanism which, in such a scenario, suspends the consumer task and resumes it only when data is ready.
– An I/O-aware application (**"IO"**), as in Figure 3(c). A single task is running on every system processor. These tasks do not communicate with each other, and perform independent computation. However, at random times, a system I/O device sends an interrupt to all of the cores to signal data availability. In response to this signal, all of the processors execute an interrupt handler routine, which moves data blocks across the system interconnect. When such handling is completed, tasks resume their normal operation.

Even in these three experimental applications, the effort required to accurately capture the interrupt propagation (and therefore the synchronization schemes) is not trivial.

The applications described above are timing-sensitive. However, within the single task, the overall performed computation does not change depending on the order of arrival of external events, and data dependencies can be captured. Only the amount of computation between each pair of events can vary. Should an environment constraint not be satisfied, tasks always enter some form of suspension, albeit in very different manners in each of the three examples. So, while an execution trace of these benchmarks shows varying traffic patterns depending on external timings, the major computation blocks are still recognizable.

Even though tasks with even more timing-dependent behaviour do exist, modeling such tasks requires an intra-task notion of context switching, which we omit here. It is worth stressing that, though not all interrupt-driven behaviours are represented, the applications we try to analyze here are definitely representative of a vast class of computation. The model we will propose can capture all such dynamics with proper insight on the mechanics of the applications and the OS.

Table 1. OCP master TG instruction set.

Instruction	Size (Words)	Description
OCP Instructions		
`Read(AddrReg)`	1	Read from an address
`Write(AddrReg, DataReg)`	1	Write to an address
`BurstRead(AddrReg, CountReg)`	1	Burst read an address set
`BurstWrite(AddrReg, DataReg, CountReg)`	1	Burst write an address set
Other Instructions		
`SetRegister(reg, value)`	2	Set register (load immediate)
`If(arg1, arg2, operand)`	2	Branch on condition
`Jump(label)`	1	Branch direct
`Idle(counter)`	1	Wait for given no of cycles

4 Support for Application Flow Replication

In this section, we describe (i) an instruction set which is capable of replicating the traffic patterns generated by an IP core, (ii) an implementation of it by means of a Traffic Generator Instruction Set Simulator (TG ISS), and (iii) an example program written to exploit TG capabilities. The whole approach significantly extends [10] to support interrupts and task switching.

The TG has an OCP master interface, and it can emulate IP cores running one or multiple tasks with and without OS. The TG is able to issue a sequence of communication transactions separated by idle wait periods, based on the programmed flow control conditions. In order to handle interrupts and other synchronization events, it is *reactive*: for example, if necessary, it is able to switch between tasks upon notification. The TG is implemented as a non-pipelined processor with a very simple instruction set, as listed in Table 1. The processor has an instruction memory and a register file for each task, but no data memory. The instruction set consists of a group of instructions which issue OCP transactions and a group of instructions allowing the programming of conditional sequencing and parameterized waits. Within the register file, some registers are designated as special purpose for flow control management; their usage is described in Table 2. The rest are general purpose registers, and their number can be configured.

Of the interrupt-related registers, `IntrpMaskReg` can be used to mask critical sections of the TG program from interrupts. As seen in Section 3, different applications require different responses to interrupt events. For example, in **IO** modeling, the main task is always interruptible, while once in the OS's interrupt handling routine, additional (nested) interrupts should be disabled. In **pipe** modeling, the interrupt handling is more specialized: interrupts are only enabled after the task has suspended, while they are masked during normal operation. `IntrpReg` holds the base location of the interrupt handling code within the TG program. `SWIntrpReg` allows the TG program to assert "software interrupts", to which the TG model will react with jumps to different parts of the program. Software interrupts are managed internally by the TG model. In contrast, hardware interrupts are routed through external wires from the NoC, and are

Table 2. TG Special Registers.

Special Register	Usage
Interrupt Registers	
`IntrpMaskReg`	Masks or unmasks interrupts
`IntrpReg`	Stores a backup of the program counter
`SWIntrpReg`	Sends a software interrupt from within the program
Other Registers	
`ThrdIDReg`	Stores the ID of the current task
`RDReg`	Stores the data value returned by the `Read(AddrReg)` instruction
`RtnReg`	Stores a jump target location

available on the sideband signals (`SInterrupt`) of the OCP interface. `ThrdIDReg`, `RDReg` and `RtnReg` provide support for specific flow control functions.

Within the TG ISS, by maintaining copies of the Program Counter (PC) and register file associated with each subtask, the context switching upon an interrupt event can be realized. Upon interrupt notification, the values of the PC and register file of the interrupted task are saved, the PC is updated with a value read from the special register `IntrpReg`, and the register file values for the designated task are loaded. It is afterwards possible to safely exit from the interrupt routine and resume a suspended task by jumping to the backup value of the source PC and reloading the backup of the register file.

Let us now consider an example of a TG program. In Figure 4, a program to model the **IO** application is sketched; the interrupt handling routine is coded together with the task itself. The TG program starts with a header describing the type of core and its identifier. The next few statements express initialization of the register file. The PC is increasing by either one or two locations along the trace; this is because some of the opcodes in Table 1, namely `SetRegister` and `If`, require longer operands and therefore fill two program slots. The main body of the TG program is composed of sequences of bus reads and writes, interleaved with register accesses (mostly to set up transaction address and data). Flow control instructions are inserted where appropriate. The interrupt handling routine is located at PC 37; this base address is stored in `IntrptReg`, which is initialized at PC 2. Within the interrupt routine, which is the critical section of the flow, interrupts are disabled. Upon a hardware interrupt event, the TG swaps the content of `IntrptReg` with that of PC. The TG program then executes any OS- or programmer-driven interrupt instructions, including transactions over the communication architecture. At the end of the flow, a software interrupt is triggered to restore the PC to the previously interrupted location (retrieved from `IntrptReg`). The flow thus mimics Figure 3(c).

5 Coding TG Programs

Depending on IP model availability to the designer, different ways exist to write TG programs which best represent the desired type of traffic.

```
MASTER[<coreID>]
; Initializations
  ...
  REGISTER IntrpMaskReg 0          ; Mask HW interrupts
  ...
BEGIN                              ; Comments                PC
  SetRegister(IntrpMaskReg, 1)     ; Unmask HW interrupts    0
  SetRegister(IntrptReg, 37)       ; Int handler is at PC 37 2
  Idle(10)                         ; Idle for 10 cycles      4
  ...
  SetRegister(AddrReg, 2)          ; Normal flow             10
  SetRegister(DataReg, 1)          ;                         12
  Write(AddrReg, DataReg)          ;                         14
  ...
  Jump(myPRGM)                     ; Jump to PC 58           36
; Continue to normal flow
; Start Interrupt Handling
IRC SetRegister(IntrpMaskReg, 0)   ; Mask HW Interrupts      37
  SetRegister(AddrReg, 23)         ;                         39
  SetRegister(DataReg, 1)          ;                         41
  Write(AddrReg, DataReg)          ;                         43
  ...
  SetRegister(IntrpMaskReg, 1)     ; Unmask HW interrupts    54
  SetRegister(SWIntrpReg, 1)       ; Trigger SW interrupt    56
; End Interrupt Handling
; Normal Application Flow
myPRGM SetRegister(AddrReg, 11)    ;                         58
  Read(AddrReg)                    ;                         60
  ...
END                                ;                         124
```

Fig. 4. IO TG Program.

5.1 Trace Parsing

In this scenario, availability of a pre-existing model for the IP under study is assumed. In this case, the approach for TG program generation goes through two steps:

- A reference simulation is performed by using the available IP model, even plugged into a different SoC platform from the target one. An execution trace is collected.
- The trace is parsed with an off-line tool. The output of the tool is the desired TG program.

In this approach, the IP core to be modeled by the TG is actually available in advance. Nevertheless, there is a rationale for still wanting to deploy the TG. The TG-based flow might provide a quick functional yet cycle-accurate port of the IP model to a SoC platform, in which, for whatever reason (*e.g.* licensing or technical issues), the

IP model might not be directly or immediately suitable for integration. Moreover, the TG device allows for a somewhat faster system simulation speed, which is valuable in the design space exploration stage.

The off-line parsing tool must of course have some notion about the traced application in order to correctly analyze and rearrange execution traces into TG programs. While this effort is not trivial, we will show its feasibility by presenting a complete validated cycle-accurate flow in Section 6.

5.2 Trace Parsing and Editing

In a related scenario, an IP model might be available, but it may differ under some respect from the IP that will eventually be deployed in the SoC device. The designer may then follow a route similar to the one outlined above. However, an additional off-line postprocessing tool might be interposed to edit the reference trace so that it more closely resembles that of the target IP. Some examples of the editing steps which are possible include:

- Removing or adding bus transactions to model a more or less efficient cache subsystem
- Removing or adding bus transactions to model a more or less comprehensive target Instruction Set Architecture (ISA)
- Altering the spacing among bus transactions to reflect different pipeline designs or timing properties
- Grouping or ungrouping bus accesses to reflect write-back *vs.* write-through cache policies

The effort required to automate these kinds of trace alterations is expected to be quite low, although the alterations themselves are very dependent on the differences among the pre-existing and the final IP model. It is certainly reasonable to expect that the coding time will be substantially than that required to develop or refine the target IP model, thus allowing for earlier exploration of the interconnect design space.

In this scenario, overall cycle accuracy with respect to the eventual system is of course not guaranteed. However, the TG 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.

5.3 Direct Development

Of course, TG programs can be written from scratch. In this case, the flexible TG instruction set allows 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 4 and [10]). Additionally, the application chunks enclosed within synchronization points can quickly be rendered by exploiting the flexible loop structures provided by the TG ISS, thus

```
MCmd WR MAddr 0x01bedfb0 MData 0x00015958 MBurstSingleReq 0
        MBurstSeq INCR 0x4 MBurstLength 1 Time 6860265
SCmdAccept Time 6860295
SInterrupt SFlag 0x00000001 Time 6860310
MFlag Time 6860310
MCmd WR MAddr 0x010b48dc MData 0x00000008 MBurst SingleReq 0
        MBurstSeq INCR 0x4 MBurstLength 1 Time 6860375
SCmdAccept Time 6860385
MCmd RD MAddr 0x0100acb0 MBurstSingleReq 1
        MBurstSeq INCR 0x4 MBurstLength 4 Time 6860720
SCmdAccept Time 6860730
Resp Data 0xe5901000 Time 6860760
Resp Data 0xe2411001 Time 6860780
Resp Data 0xe5801000 Time 6860800
Resp Data 0xe14f0000 Time 6860820
MCmd WR MAddr 0x0102c040 MData 0x00000000 MBurstSingleReq 0
        MBurstSeq INCR 0x4 MBurstLength 1 Time 6860830
SCmdAccept Time 6860840
```

Fig. 5. Trace file snippet.

providing periodic traffic generation capabilities at least on par with those of traditional TG implementations. An alternate possibility, as demonstrated in [7], is using the TG as an interface between formal and simulation models in a hybrid environment. Here, the TG 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 TG flow allows for deployment in a number of situations.

6 A Test Case: A Trace-Based TG Deployment Flow

To test TG accuracy and viability, we set up a validation flow following the outline described in Section 5.1. First, the user performs a reference simulation of the target applications where all IP cores are simulated using bit- and cycle-true models to collect traces from the cores' OCP interfaces. Figure 5 shows a snippet of trace file. It contains the communication event type (read, write or interrupt), its response(s), and its timestamp. Subsequently, these traces are converted into corresponding TG programs by a *translator*. Finally, a custom assembler is used to convert the symbolic TG program into a binary image which can be loaded into the TG instruction memory and executed. The trace to TG program conversion process is fully automated and the time taken for this process is nominal ([10]). The validation of the TG flow is achieved by coupling the TG with the same interconnect used for tracing with IP cores, and checking the accuracy of the resulting IP core emulation. Experimental results will be shown in Section 7.

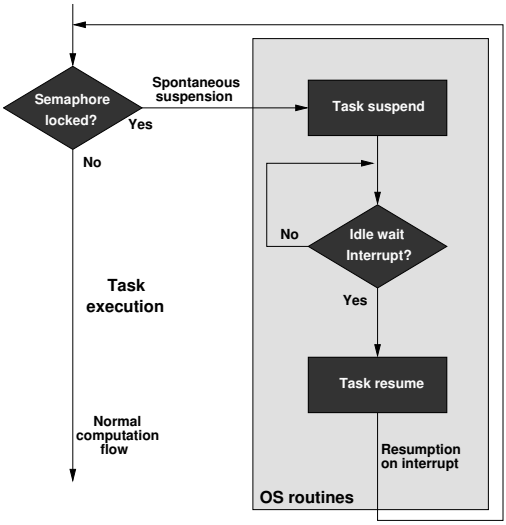

Fig. 6. Application flow, on any single core, of **pipe**.

Even though modeling an application in presence of interrupt handling is not straightforward, we show an automated flow capable of capturing many synchronization behaviours which are typical of complex systems. The designer does not need to handle them manually. Algorithms to detect such behaviours in the applications of Section 3 are shown next.

Depending on the target application, one or more of the following pieces of information can be extracted about interrupt handling from the trace file to help the translator tool:

– the time when interrupt events occur,
– the end of an interrupt handling routine,
– the spontaneous suspension waiting for an interrupt in idle state.

The amount of annotations that can be extracted reflects the degree of access the programmer has to the interrupt routine and to the OS internals. In the **IO** test case, the interrupt handling is likely to be part of the functionality of a custom device driver, and thus we assume that the programmer has full access to both the code of the application and of the interrupt handler. Therefore, trace files contain the time of occurrence of the interrupt event; custom markers (*i.e.* dummy memory accesses to specific locations) can be appended by the programmer at the end of the interrupt handling routine. The transactions within these bounds can be detected as interrupt handling code and be encapsulated as such in the TG program.

In the **pipe** scenario, the task is interacting with the OS internals by voluntarily suspending should certain conditions be true (*i.e.* finding a semaphore locked). Additionally, the task negotiates with the OS to be resumed upon interrupt receipt. The task may also want to ignore an interrupt in the following condition: it is possible that the upstream producer, or the downstream consumer, notifies availability of data or buffer space before the actual need for such resources, because the current task is still

busy with previous internal processing. Despite the complex interaction, usually the synchronization functionality required by **pipe** can be achieved by properly using OS APIs, without direct access to the interrupt handler code, whose exit point is therefore assumed to be not accessible by 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. A TG program can thus mimic the flow shown in Figure 3(b), first by reading the semaphore location, and then by choosing to continue or suspend depending on the lock. Upon resumption by hardware interrupt, a final (re-)check of the semaphore unlock can be done to ensure safe task operation. Figure 6 shows the equivalent flow. In the TG program, hardware interrupts are used to wake up from the suspension state within OS routines, while software interrupts redirect the execution flow towards the main task. Note that `IntrpMaskReg` is set to the masked state for the regular program and OS execution, and is only unmasked within the suspended state.

In the **task** benchmark, the interrupt handler is typically completely out of the programmer's control, as it is tied to the OS scheduling code. The tasks are not explicitly notified upon the receipt of an interrupt, and are just suspended and resumed by the OS. Therefore, trace files are annotated only with the time of occurrence of interrupt events. The TG execution toggles among tasks upon these interrupts. This is not very different from **IO**, but, since it is assumed for the programmer to be impossible to explicitly tag the handler exit point with a custom flag, the interrupt handling routine is merged with a stage of the next scheduled task because the translator tool has no way to detect this jump. Additionally, control is never spontaneously released by means of software interrupts: the previously active task is only resumed upon arrival of a hardware interrupt. The TG ISS automatically supports context switching, as described in Section 4, with multiple register sets.

Once critical points within the trace file are recognized, the translator tool accordingly inserts interrupt handling routines into the TG programs by using the TG flow control instructions described in Section 4. The above mentioned issues in flow recognition within the traces (*e.g.* interrupt handler code being captured as a part of the instructions of the next task) introduce some minor inaccuracies, which will be quantified in Section 7.

7 Experimental Results

We coded the three test cases mentioned in Section 3 as tasks running on top of an operating system and we simulated them within the MPARM framework. Each was tested with two (2P), four (4P) and six (6P) system processors. For **task** and **IO**, we devoted one of the system cores to the generation of interrupts, emulating the role of a timer or an IO device; this processor is not generating any other traffic on the bus, and is just idling between interrupt generation events. The **pipe** benchmark does not need this, since interrupts are directly triggered by the same tasks which perform the computation. Subsequently, we applied the flow described in Section 6 as one of the ways to get TG programs.

Table 3. TG vs. ARM performance with AMBA.

Benchmark	# IPs	ARM				TG			
		Execution Cycles	Reads	Writes	Sim Time (s)	Execution Cycles	Read	Writes	Sim Time (s)
task	2	5864410	24163	142529	109	5863463	24163	142532	48
	4	6357457	53618	362000	205	6353359	53627	362020	92
	6	7029779	83134	582383	299	6966958	82351	578375	140
pipe	2	621954	16809	48268	9	627326	16812	48267	5
	4	961581	34300	98143	20	980000	34305	98143	13
	6	1390443	51251	148242	37	1417000	51261	148241	27
IO	2	1754773	23999	78379	30	1749258	23999	78379	15
	4	2118506	53491	180169	58	2117514	53515	180169	31
	6	2647029	82966	281967	93	2647071	82989	281942	53

Table 4. Relative Error and Speedups.

Benchmark	# IPs	Relative Error			Speedup (x)
		Execution Cycles	Reads	Writes	
task	2	0.02%	0.00%	0.00%	2.27
	4	0.06%	0.02%	0.01%	2.22
	6	0.89%	0.94%	0.69%	2.13
pipe	2	0.86%	0.02%	0.00%	1.8
	4	1.92%	0.01%	0.00%	1.53
	6	1.91%	0.02%	0.00%	1.37
IO	2	0.31%	0.00%	0.00%	2
	4	0.05%	0.04%	0.00%	1.87
	6	0.00%	0.03%	0.01%	1.75

Table 3 shows statistics for experiments carried out within MPARM, both with TG-injected traffic and with the original ARM cores. The figures express:

- the number of clock cycles required to complete a benchmark run, from the boot to the end of the execution of the last processor;
- the amount of bus accesses done by a core to perform a read;
- the amount of bus accesses done by a core to perform a write;
- the number of seconds taken by the simulator to complete a benchmark run.

Table 4 shows the relative error in execution time and number of bus accesses when contrasting the original execution on ARM cores and that on traffic generators, and simulation speedup values. Figure 7 depicts the accuracy of our modeling scheme, by plotting the relative error values. Errors are due to an improper modeling of the application under test, which misplaces some bus accesses done by the real cores when mapping them onto a TG program. For example, this may happen if a bus access belonging to an interrupt handler is mistakenly assigned to the main application task when detecting the application flow within the execution trace. In turn, such misplacements result in skews of bus transactions and arbitrations, which potentially propagate

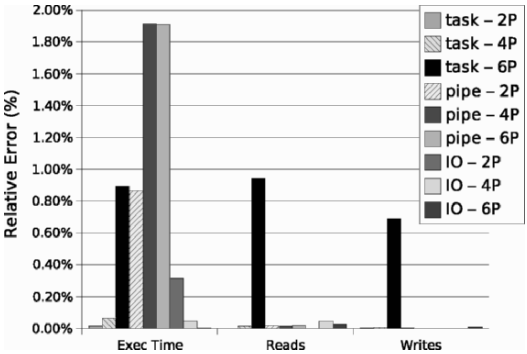

Fig. 7. Accuracy of the execution on TGs *vs.* on the original ARM cores.

across the benchmark run, therefore causing a difference in the final execution cycle count. Such skews can also affect the amount of actual bus accesses, for example whenever a semaphore polling has to be performed and the timing of the bus access for the semaphore release is shifted in time.

The plot shows a good match between ARM and TG runs. The typical error, both in execution time and bus accesses, is below 2%, resulting in a faithful reproduction of the original execution flow and traffic patterns. The near-matching amount of read and write accesses proves the role of the TG as a powerful design tool to mimic complex application behaviour in replacement of a real IP core. Additionally, the correctness of our TG program translation is validated. Some mismatches can be observed especially in the execution time for the **pipe** benchmark. These are due to minor issues in properly pinpointing single sections of internal OS code in the execution trace.

Figure 8 reports the simulation time speedup achieved as a side advantage when running the benchmark code on TGs as opposed to ARM ISSs[1]. A nominal gain of 1.37x to 2.27x can be observed. The **task** and **IO** benchmarks exhibit a higher improvement due the presence of an IP core which is idle for most of the time, in the time lapses between interrupt injections. In addition, the **pipe** benchmark is at a disadvantage due to a higher bus utilization (with six processors, 78% against 63% for **IO** and 38% for **task**), which shifts simulation time emphasis upon the interconnect model. This also explains why **task** has the best speedup figures.

In terms of scalability, while it might be expected that replacing increasing numbers of IP cores with traffic generators should yield increasingly better performance, this is not always true; while the absolute gain is present and increasing, the relative speedup can often decrease. The explanation for this is that, with more cores attached to the system bus, congestion becomes an issue and more core cycles are spent waiting for bus arbitration. In this case, there is no simulation time advantage in replacing full-blown ISSs with traffic generators.

[1] Benchmarks taken on a multiprocessor Xeon® 1.5 GHz with 12 GB of RAM, thus eliminating any disk swapping or loading effect. Time measurements were taken by averaging over multiple runs.

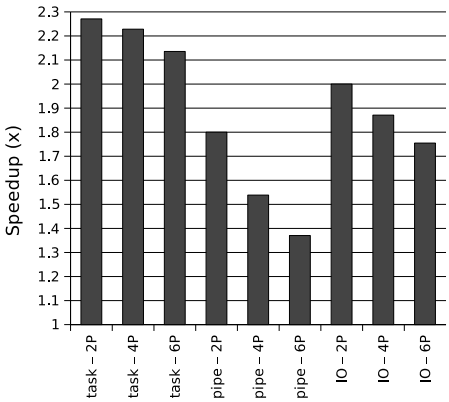

Fig. 8. Simulation speedup when replacing the original ARM cores with TGs.

8 Conclusions

Experimental results proved the viability of a modeling approach which decouples simulation and optimization of IP cores and of interconnect fabrics. Even when tested under complex synchronization scenarios, including asynchronous interrupts involving OS interaction in a multiprocessor environment, the proposed instruction set is able to reproduce IP traffic with full capability to express the application flow. Multiple ways to write programs for this architecture are suggested, and a thorough analysis of one of them is presented. The accuracy of a simulation device providing an implementation of said instruction set is validated in a cycle-true environment by benchmarking multiple applications, additionally achieving a nominal but noticeable simulation speedup.

Future work will revolve around improving the accuracy of our flow, by more clearly detecting sections of input traces and rendering them as completely separate tasks within TG programs. We also plan on carefully studying the impact of changes in modeled traffic onto the interconnect congestion and therefore on communication latency.

9 Acknowledgments

The work of Shankar Mahadevan is partially funded by SoC-Mobinet and ARTIST II. The work of Federico Angiolini is partially funded by ST Microelectronics and by the SRC program.

References

1. The Real-Time Operating System for Multiprocessor Systems. http://www.rtems.com.
2. Open Core Protocol Specification, Release 2.0, 2003.

3. F. Angiolini, S. Mahadevan, J. Madsen, L. Benini, and J. Sparsø. Realistically rendering SoC traffic patterns with interrupt awareness. In *IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, September 2005.
4. L. Cai and D. Gajski. Transaction level modeling in system level design. CECS technical report 03-10, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, March 2003.
5. 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 *Proceedings of the 42th Design Automation Conference (DAC)*, pages 42–47, June 2003.
6. T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
7. S. Kuenzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proceedings of Design, Automation and Testing in Europe Conference 2006 (DATE)*, pages 236–242. IEEE, March 2006.
8. K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of System-on-Chip communication architectures. In *Proceedings of the 14th International Conference on VLSI Design*, pages 29–35, 2001.
9. M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2004.
10. 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 *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE)*, pages 780–785. IEEE, March 2005.
11. 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 *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE)*. IEEE, March 2003.
12. S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of 38th Design Automation Conference (DAC)*, pages 113–118. ACM, 2004.
13. L. Schaelicke, A. Davis, and S. A. McKee. Profiling IO interrupts in modern architectures. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2000.
14. 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 *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 667 – 672, June 2001.
15. W. Wolf. *Computers as Components:Principles of Embedded Computing System Design*, chapter 3. Morgan Kaufmann, 2001.