A Network Traffic Generator Model for Fast Network-on-Chip Simulation

Mahadevan, Shankar; Angiolini, Frederico; Storgaard, Michael; Olsen, R.; Sparsø, Jens; Madsen, Jan

Published in:
Design, Automation and Test in Europe

Link to article, DOI:
10.1109/DATE.2005.22

Publication date:
2005

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

Citation (APA):
Mahadevan, S., Angiolini, F., Storgaard, M., Olsen, R., Sparsø, J., & Madsen, J. (2005). A Network Traffic Generator Model for Fast Network-on-Chip Simulation. In *Design, Automation and Test in Europe* (pp. 780-785) https://doi.org/10.1109/DATE.2005.22

A Network Traffic Generator Model for Fast Network-on-Chip Simulation

Shankar Mahadevan[†] Federico Angiolini[‡] Michael Storgaard[†] Rasmus Grøndahl Olsen[†]
Jens Sparsø[†] Jan Madsen[†]

[†] Informatics and Mathematical Modelling (IMM)
Technical University of Denmark (DTU)
Richard Petersens Plads, 2800 Lyngby, Denmark
e-mail: {sm, -,-, jsp, jan}@imm.dtu.dk

[‡] Dipartimento di Elettronica, Informatica e Sistemistica (DEIS)
University of Bologna
Viale Risorgimento, 2 40136 Bologna, Italy
e-mail: {fangiolini}@deis.unibo.it

Abstract

For Systems-on-Chip (SoCs) development, a predominant part of the design time is the simulation time. Performance evaluation and design space exploration of such systems in bit- and cycle-true fashion is becoming prohibitive. We propose a traffic generation (TG) model that provides a fast and effective Network-on-Chip (NoC) development and debugging environment. By capturing the type and the timestamp of communication events at the boundary of an IP core in a reference environment, the TG can subsequently emulate the core's communication behavior in different environments. Access patterns and resource contention in a system are dependent on the interconnect architecture, and our TG is designed to capture the resulting reactiveness. The regenerated traffic, which represents a realistic workload, can thus be used to undertake faster architectural exploration of interconnection alternatives, effectively decoupling simulation of IP cores and of interconnect fabrics. The results with the TG on an AMBA interconnect show a simulation time speedup above a factor of 2 over a complete system simulation, with close to 100% accuracy.

1 Introduction

An important step in the design of a complex System-on-Chip is to select the optimal architecture for the on-chip network (NoC). In order to do so, it is imperative to analyze and understand network traffic patterns through simulation. This can be accomplished at various stages in the design flow, from abstract transaction level models (TLM) to bit- and cycle-true models. In many cases, only the most detailed models prove capable of capturing important aspects of communication performance, e.g. the latency associated with resource contention. The obvious drawback of these approaches is slower simulation speed.

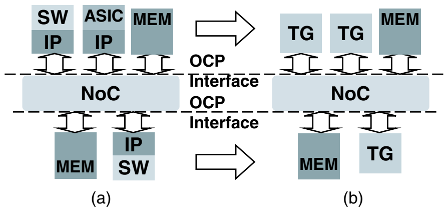


Figure 1. Simulation Environment with bit- and cycle-true: (a) IP-cores, (b) TG model.

In this paper, we focus on enabling the exploration of different NoC architectures at the bit- and cycle-true level by increasing the speed of the complete SoC simulation. This is a key advantage, since architectural exploration typically involves carrying out the same set of simulations for each design alternative, and simulations may consist of millions of clock cycles each.

We assume as a requirement the availability of a reference SoC design, consisting of IP cores and of a NoC, and of an application partitioned and compiled onto the various IP cores. This application might either be software executing on programmable IP cores, or code synthesized into dedicated hardware. This reference system will be used to collect an initial trace of the IP cores' behavior. In order to increase simulation speed for subsequent design space exploration, we propose to replace the IP cores with Traffic Generators (TG) which emulate their communication at the interface with the network, as illustrated in Figure 1.

The goal is to perform only one reference simulation using bit- and cycle-true simulation models of the IP cores running the target application, and to speed up subsequent variants of that simulation using traffic generators coupled with accurate models of the alternative interconnects only. While the internal processing of IP cores does not need thorough replication by the generators, and can often be modeled by waiting for an amount of cycles between network transactions, the unpredictability of network latency of different NoC architectures may lead to changes in the number and relative ordering of transactions. Thus, traffic generators should have at least some reactive capabilities, as will be explained in Section 3.

In order to capture reactive behavior, we propose a TG implementation as a very simple instruction set processor. Our approach is significantly different from a purely behavioral encapsulation of application code into a simulation device, in analogy with TLM modeling. The TG model we propose is aimed 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 and of latencies between accesses, allowing for cycle-true simulations. At the same time, this approach allows a straightforward path towards deployment of the TG device on a silicon NoC test chip.

To evaluate the TG concept, we have integrated the proposed TG model into MPARM [8], a homogeneous multiprocessor SoC simulation platform, which provides a bit- and cycle-true SoC simulation environment. The current version of MPARM supports several NoC architectures, e.g. AMBA [8], STBus and the ×pipes [3], and leverages ARMv7 processors as IP cores. The use of the OCP [1] protocol at the interfaces between the cores and the interconnect allows for easy exchange of IP cores for TGs, as indicated in Figure 1.

The rest of the paper is organized as follows. Section 2 introduces related work, and is followed by a discussion of the requirements for modelling traffic patterns in Section 3. Section 4 details the TG implementation, and Section 5 describes how communication traces are extracted and turned into programs executing on the TG. Section 6 presents initial simulation results which show the potential of our TG approach. Finally, Section 7 provides conclusions.

2 Related Work

The use of traffic generators to speed up simulation is not new, and several traffic generator approaches and models have been proposed.

In [6], a stochastic model is used for NoC exploration. Traffic behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. Such distributions assume a degree of correlation within the communication transactions which is unlikely in a SoC environment. Traffic patterns in SoC systems have shown to be reactive and bursty [2, 7]. 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, 9, 10, 11]. In [9, 10], 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 [10] 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, thus intra-transaction visibility is traded-off for simulation speedup. While modeling the entire system at higher abstraction i.e. TLM, both [9] and [10] present a methodology for preserving accuracy with gain in simulation speed.

We would like to underline that our approach is dual with respect to TLM. While transaction-level models usually represent interconnects as a collection of available services and emphasize local processing on IP cores, the platform we describe is composed of accurate models for the interconnect, while processing resources are abstracted away. Simulation speed is gained like in TLM models, but the purpose of this gain is enabling accurate assessment of interconnect performance, not of core or application performance. The above methods are suitable for feature exploration once the NoC architecture has been chosen, but are not thought for NoC exploration itself.

3 Traffic Modeling Requirements

The generation of a traffic pattern emulating that of a real IP core can be faced at varying degrees of accuracy.

At the most basic level, a trace with timestamps can be collected in the reference system and then be independently replayed, an approach that we might call "cloning". This approach is clearly inadequate when the variance of network latency is taken into account; whenever a transaction is delayed, either due to hardware design or congestion, the effect should propagate to subsequent transactions, which would also be delayed in real systems. A simple example of such critical blocking is a cache refill request.

This observation leads to the deployment of "timeshifting" traffic generators: adjacent 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. This implicitly means that the trace collection mechanism must include not only timestamps for processor-generated commands, but also for network responses. However, even this model fails when multi-core systems come under scrutiny: the arbitration for resources in such designs is timing-, and thus architecture-, dependent. Therefore, very different transaction patterns may be observed as a function of the chosen interconnection design. To make an example, checks for a shared resource done by polling generate different amounts of traffic depending on the relative ordering of accesses to the resource.

As a consequence, the need for "reactive" TG models is justified. Such models must have some knowledge about the system architecture and about the application behavior to correctly generate (and not just duplicate) traffic patterns

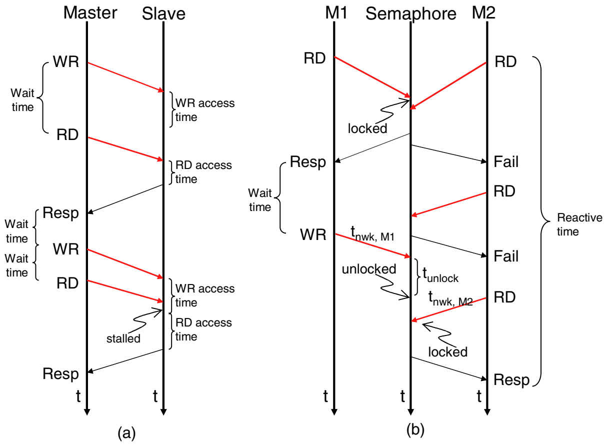


Figure 2. Two typical MPARM transactions.

across different underlying networks. A TG should be able to mimic the behavior of an IP core even when facing unpredictable network performance, e.g. due to resource contention, packet collisions, arbitration and routing policies.

To illustrate the requirements driving the development of our TG model, we will now describe how two typical transactions occurring in the MPARM modelling and simulation environment can be reproduced. MPARM features in-order, single-pipeline ARM cores as system masters and two types of memory as system slaves: private (only accessible by one master) or shared (visible by all masters in the system). Figure 2 shows examples of the two types of communication: (a) processor-initiated communication towards an exclusively owned slave peripheral, and (b) processor-initiated communication towards a system-shared slave peripheral. We call network latency (t_{nwk}) the time taken for the communication to traverse from the master OCP interface to the slave OCP interface and vice versa; this latency depends both on the chosen architecture and on the network congestion at the time of communication.

In Figure 2(a), the first two master transactions are a write (WR) and a read (RD). The time to service the WR transaction, which is a posted write, is just the network latency plus the slave access time. The RD, which in MPARM uses blocking semantics, pays an additional penalty because the response has to make its way back to the master. From the TG point of view, this pattern is easily recordable: network latency and slave access time are unimportant factors, and the essential point to capture is just the delay between WR assertion and RD assertion, and between RD response and the following command. In a subsequent simulation with traffic generators replacing cores, these delays will be modeled by explicit idle waits in the TG, while the network latency will be dependent on the NoC model to simulate. In the next set of transactions, where a RD closely follows a WR, the RD command reaches the slave before the latter has finished servicing the WR, and is thus stalled at the slave interface. This stalling behavior does not need to be explicitly captured in a TG model, since, from a processor perspective, it simply appears to be part of the slave response time. This simplistic example of a master accessing a private slave proves that if the type and the timestamp of the communication events are captured, the behaviour of the master can be emulated via non-preemptive sequential communication transactions interleaved with an appropriate amount of idle wait cycles.

In Figure 2(b), two master devices (M1 and M2) attempt to gain access to a single hardware semaphore. M1 arrives first and locks the resource; the attempt by M2 thus fails. In MPARM, semaphore checking is performed by polling, i.e. M2 regularly issues read events until eventually the semaphore is granted to it. Since the transactions occur over a shared network fabric, the unlock event (WR) issued by M1 and the success of the next request (RD) event by M2 are dependent. Only if the M2 RD event is issued at least $t_{\text{nwk,M1}} + t_{\text{unlock,S}} - t_{\text{nwk,M2}}$ after the unlocking by M1, then M2 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 OCP interfaces of M1 and M2. This is the *reactive behavior* that needs to be captured by the TG model: both M1 and M2 need to react to accommodate the network latency. Thus, the simplistic model which could be applied to transactions towards a privately owned slave now needs to be extended with additional information about the master process execution, about system properties, and about input/output data. In detail, the TG must be able to recognize polling accesses (i.e. a knowledge of what addressing ranges represent pollable resources) and must add support for recording of actual data transfers (e.g., writing a "1" or a "0" to a shared memory location might be the difference between locking or unlocking a resource).

We take the above discussion as a requirement to implement accurate TG models. The examples in Figure 2 demonstrate that traces collected at the IP-NoC interface are sufficient to accurately reproduce the IP's communication, provided that the reactive behavior of the master IP cores is taken into account. These traces should collect sequences of communication transactions, comprising of requests and responses, separated by time intervals with no communication, i.e. idle time. A simulation of the entire system should produce several traces, one per IP core interface.

4 Implementation of the traffic generators

In this section we describe in some detail the implementation of our traffic generators. As mentioned before, our proposed TG model is designed as a simulation tool, but allows future deployment as a hardware device. Within the simulation environment for NoC exploration, the emphasis is on simulation speedup, while within a hardware instance the emphasis is on ease of (re)programmability and a small silicon footprint in order to support implementation of test chips containing NOC prototypes.

Conceptually, three different TG entities might be needed: (1) A TG emulating a processor (an OCP master). This TG must be able to issue conditional sequences of traces composed of communication transactions separated by idle/wait-periods; (2) A TG emulating a shared memory (an OCP slave). This TG must contain a data structure modeling an actual shared memory (since the values read by the masters may affect the sequence of transactions seen at the master IP cores); and (3) A TG emulating a slave memory (an OCP slave). This TG must be able to respond, possibly with dummy values, to communication transactions issued by a master. Only the first is actually required for deployment in a simulation environment, which already provides its own system slaves, thus only this entity will be described in the present paper. On the other hand, it is important to notice that both slave TG modules are much simpler in design with respect to the master TG, as their logic basically just involves a small state machine to handle OCP transactions.

For the processor TG, we have implemented and modeled a multi-cycle processor with a very simple instruction set as listed in Table 1. The processor has an instruction memory and a register file, but no data memory. The instruction set consists of a group of instructions which issue OCP transactions (whose arguments are set up in registers) and a group of instructions allowing the programming of conditional sequencing and parameterized waits such that the required traces can be implemented/programmed. The process for deriving TG programs from traces obtained in the reference simulation is explained in Section 5.

5 The TG simulation flow

In order to use the traffic generators, a user must first perform a reference simulation using bit-true and cycle-true IP models. It is interesting to note that, at this stage, the interconnect does not yet need to be accurately modeled, allowing for time savings. During this simulation, traces are collected from all OCP interfaces in the system. For this purpose, the OCP interface modules within the MPARM platform (the network interfaces in the case of the ×pipes interconnect, the bus master in the case of AMBA AHB) were adapted to collect traces of OCP request and response communication events into a predefined file format (*.trc*). The address and (if any) data fields of the transactions are also observed. Trace entries are single or burst read/write transactions. Figure 3(a) shows an example trace.

The next step is to convert the traces into corresponding TG programs (*.tgp*). A translator outputs symbolic code; Figure 3(b) shows the TG program derived for traces in Figure 3(a). Finally, an assembler is used to convert the symbolic TG program into a binary image (*.bin*) which can be loaded into the TG instruction memory and executed. Execution might be within a simulation model (which is the approach presented in this paper) or in hardware on a NoC test-chip. Validation of the trace collection and processing mechanism can be achieved by collecting traces with IP cores running on different interconnects, and verifying the resulting *.tgp* and *.bin* programs to match. The conversion process is fully automated and the time taken for this process is discussed in Section 6.

Instructions	**Description**
OCP Instructions:	
Read(addr)	Read from an address
Write(addr, data)	Write to an address
BurstRead(addr, count)	Burst read a range of addr.
BurstWrite(addr, data, count)	Burst write an address set
Other Instructions:	
If(arg1, arg2, operand)	Branch on condition
Jump(location)	Branch direct
SetRegister(reg, value)	Set register (load immediate)
Idle(counter)	Wait for given no of cycles

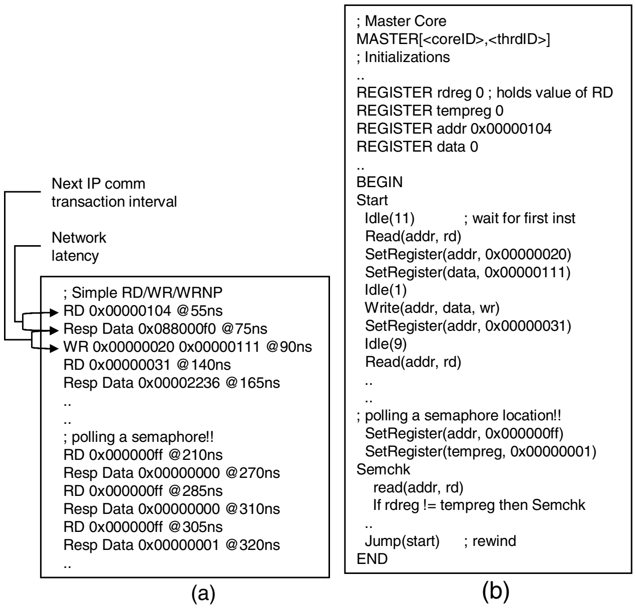
Table 1. OCP-master TG instruction set.

As seen in Figure 3(b), 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. Register `rdreg` is defined as special register where the value of RD transactions is stored.

By looking at the code in Figure 3(a), it is possible to notice that the first communication events in the trace occur at time 55ns, 75ns, and 90ns. We assume each TG cycle to take 5ns, the same as the IP core for which the trace is collected. At the beginning of the simulation, the TG has no instruction to perform until the 11th (55/5) cycle, so an `Idle` wait is observed. The trace of the RD event is followed by a response, at a time which is dependent on the network latency. The IP core is blocked until this response arrives. A WR event occurs three ((90-75)/5) cycles after the response is received; these cycles are partially spent for TG internal operations (data and address register setting), and an ensuing `Idle` wait is added to fill the gap. Then the following RD instruction is translated into the corresponding `Read` program call after 10 cycles, one of which is taken to set up the RD address. This is blocking until a response is received, 5 cycles later.

Now, consider the trace entries from time 210ns to 320ns. By identifying the address as belonging to a semaphore location and knowing the polling behaviour of the MPARM 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 unblocked semaphore. This loop effectively represents the semaphore polling behavior. All master devices attempting to access this address incorporate the same routine in their TG program, thus capturing the system dynamics.

At this stage, additional simulations can be run on a plat-



```
; Simple RD/WR/WRNP
RD 0x00000104 @55ns
Resp Data 0x088000f0 @75ns
WR 0x00000020 0x00000111 @90ns
RD 0x00000031 @140ns
Resp Data 0x00002236 @165ns
..
..
; polling a semaphore!!
RD 0x000000ff @210ns
Resp Data 0x00000000 @270ns
RD 0x000000ff @285ns
Resp Data 0x00000000 @310ns
RD 0x000000ff @305ns
Resp Data 0x00000001 @320ns
..
```

```
; Master Core
MASTER[<coreID>,<thrdID>]
; Initializations
..
REGISTER rdreg 0 ; holds value of RD
REGISTER tempreg 0
REGISTER addr 0x00000104
REGISTER data 0
..
BEGIN
Start
 Idle(11)          ; wait for first inst
 Read(addr, rd)
 SetRegister(addr, 0x00000020)
 SetRegister(data, 0x00000111)
 Idle(1)
 Write(addr, data, wr)
 SetRegister(addr, 0x00000031)
 Idle(9)
 Read(addr, rd)
 ..
 ..
; polling a semaphore location!!
 SetRegister(addr, 0x000000ff)
 SetRegister(tempreg, 0x00000001)
Semchk
  read(addr, rd)
  If rdreg != tempreg then Semchk
 ..
 Jump(start)      ; rewind
END
```

Figure 3. (a) MPARM Trace to (b) TG Program.

form with traffic generators and a variety of interconnect fabrics, thereby evaluating performance of NoC design alternatives. Compared with the reference setup, where the interconnect fabric could be modeled at a high level, the target NoC should now be simulated at the cycle- and bit-true level to carefully assess its performance. Validation of the TG model can be achieved by coupling the TG with the same interconnect used for tracing with IP cores, and checking the accuracy of the IP core emulation. Results for this validation, and for tests on different interconnects than the reference one, will be presented in the next Section.

6 Results

We simulated within the MPARM framework, using the AMBA NoC, and four benchmarks. The first benchmark was a single-processor application (SP matrix manipulation), with the purpose of assessing accuracy and speedup in the simplest environment. The second benchmark (Cacheloop) was a test performing idle loops within the processors' cache, and only minimal bus interaction; this allowed an assessment of the speedup provided by the TG model when scaling the number of processors in the system up to twelve. Finally, the remaining two benchmarks (MP matrix manipulation and DES encryption/decryption) were multiprocessor tests stressing synchronization and resource contention with traffic patterns as discussed in Section 3, and were used mainly to ascertain the accuracy of the whole design flow when stressed by complex transactions.

#IPs	Cumulative Execution Time			Simulation Time		
	ARM	TG	Error	ARM	TG	Gain
SP matrix:						
1P	6610680	6610659	0.00%	73 s	34 s	2.15x
Cacheloop:						
2P	2500903	2500913	0.00%	47 s	14 s	3.36x
4P	2501760	2501701	0.00%	87 s	22 s	3.95x
6P	2502558	2502640	0.00%	127 s	29 s	4.38x
8P	2503404	2503522	0.00%	163 s	37 s	4.41x
10P	2504250	2504404	0.01%	197 s	42 s	4.69x
12P	2505096	2505286	0.01%	239 s	51 s	4.69x
MP matrix:						
2P	3276505	3276030	0.01%	66 s	25 s	2.64x
4P	3528038	3530759	0.08%	128 s	42 s	3.05x
6P	3691454	3697854	0.17%	195 s	61 s	3.20x
8P	3997878	4058812	1.52%	260 s	82 s	3.17x
10P	4881007	4902806	0.45%	334 s	106 s	3.15x
12P	5901290	5901131	0.00%	432 s	143 s	3.02x
DES:						
3P	978080	980098	0.21%	26 s	10 s	2.60x
4P	1054839	1057944	0.29%	34 s	11 s	3.09x
6P	1491570	1492274	0.05%	53 s	20 s	2.65x
8P	1959755	1960575	0.04%	73 s	30 s	2.43x
10P	2441026	2441743	0.03%	95 s	42 s	2.26x
12P	2927359	2927218	0.00%	125 s	62 s	2.02x

Table 2. TG vs. ARM performance with AMBA.

In the first experiment we aimed at validating the trace collection/processing environment. We ran the same benchmarks over AMBA and ×pipes, noticing very different execution times due to different latency and scalability features. However, after translation, a check across *.tgp* programs showed no difference at all. This result demonstrates the feasibility of an approach which decouples simulation of the IP cores and of the underlying interconnect fabric.

Table 2 summarizes the results of simulations done on the AMBA AHB interconnect with ARM processors and then with TGs. The left columns report the number of simulated cycles, while the right ones illustrate simulation time[1]. The column "Error" is a measure of the accuracy of replacing IP cores with TGs, based upon the difference in simulated cycles, while the column "Gain" describes the improvement in simulation time.

The table shows that replacing ARM processors with TGs yields excellent accuracy, close to 100% for small numbers of processors, while guaranteeing a speedup factor of 2 to 4. This speedup is mostly due to the drastic simplification in the amount of logic needed to generate communication transactions, compounded in small part with the elimination of any adaptation layer in the system since the TG is natively implemented with an OCP interface. This speedup compares favorably to previous work in the area (a speedup of 1.55x is reported in [10]), and must be evaluated by taking into account the fact that it involves no shift in the level of abstraction of the simulations.

[1] Benchmarks taken on a multiprocessor Xeon® 1.5 GHz with 12 GB of RAM, eliminating any disk swapping effect. Especially for benchmarks with a short duration, time measurements were taken by averaging over multiple runs and care was put in minimizing disk loading effects.

Inaccuracies in execution time can be explained as follows. In Cacheloop, and until about 6-8 processors in MP matrix and four in DES, the TG platform shows an acceptable accuracy degradation. This is due to the compounding of minimal timing mismatches caused by the conversion from traces to TG programs. When adding even more processors, however, accuracy improves again, because the AMBA bus starts to saturate, causing the processors to idle wait for bus arbitration for long amounts of time. The congestion is serious enough to dominate the effect of the timing mismatches. Since TGs cannot save simulation complexity if the replaced processors are in idle state, this is also the reason causing the speedup to get smaller with large numbers of processors in MP matrix and DES. For Cacheloop, which always executes from the local caches without any bus traffic, this phenomenon does not appear. Thus, the reduced speedup is not a property of the TG.

The impact of trace collection is small, and is incurred only once. For example, when running the MP matrix benchmark on the AMBA interconnect with four ARM processors, a plain benchmark run takes 128 s; the benchmark run with TG tracing enabled takes 147 s, and subsequent parsing and elaboration requires an additional 145 s for a 20 MB trace file[1]. Only one such iteration is needed to be able to take advantage of 2x to 4x speedups in subsequent design space exploration. Additionally, since processed TG programs are identical regardless of the reference interconnect in which raw traces were collected, such collection could be performed on top of a transactional fabric model, further reducing the impact of the reference simulation.

7 Conclusions

Experimental results prove the viability of a TG-based approach which decouples simulation of IP cores and of interconnect fabrics. Even in presence of unpredictable contention for shared resources in a multiprocessor environment, our TG model proved capable of delivering speedups in the order of 2x to 4x when run on AMBA while keeping a remarkable accuracy.

The TG model we propose provides a wide range of features, with a simple but powerful instruction set allowing for sophisticated flow control and therefore a variety of communication patterns. It is very useful for fast and accurate verification and exploration of different NoC architectures, which is the motivation of this work. While this paper was focused on simulation speedup, the TG may also be used as a flexible tool in a variety of platforms. The TG might be used in association with manually written programs to generate traffic patterns typical of IP cores still in the design phase, helping in the tuning of the communication performance between the underlying NoC and that IP core.

Our future work includes synthesis of the TG device, and support for processors allowing out-of-order transactions. Research will also include analysis of the behavior of a system in which multiple tasks run on a single processor and are dynamically scheduled by an OS, either based upon timeslices (preemptive multitasking) or upon transition to a sleep state followed by awakening on interrupt receipt. Context switching-related issues will need to be modeled or predicted.

8 Acknowledgments

The work of Shankar Mahadevan is partially funded by SoC-Mobinet, Nokia, and the Thomas B.Thrige Foundation. The work of Federico Angiolini is partially funded by ARTIST.

References

[1] Open Core Protocol Specification, Release 2.0 http://www.ocpip.org, 2003.

[2] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. In *Journal of Systems Architecture*. Elsevier, 2004.

[3] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xpipes: A latency insensitive parameterized Network-on-Chip architecture for multi-processor SoCs. In *Proceedings of 21st International Conference on Computer Design*, pages 536–539. IEEE Computer Society, 2003.

[4] 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*, pages 42–47, 2003.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] 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.

[7] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey. Communication architecture tuners: A methodology for the design of high-performance communication architectures for System-on-Chips. In *Proceedings of the 2000 Design Automation Conference, DAC'00*, pages 513–518, 2000.

[8] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004.

[9] 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 (DATE03)*. IEEE, March 2003.

[10] 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'04)*, pages 113–118. ACM, 2004.

[11] 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.