A Modular Fast Simulation Framework for Stream-Oriented MPSoC

Kai Huang, Iuliana Bacivarov, Jun Liu, Wolfgang Haid
Computer Engineering and Networks Laboratory, ETH Zürich, Switzerland
Email: {kai.huang, iuliana.bacivarov, jun.liu, wolfgang.haid}@tik.ee.ethz.ch

Abstract—**The performance estimation of complex multi-processor systems-on-chip (MPSoC) in a reasonable amount of time and with a good accuracy becomes more and more challenging due to the increasing number of embedded components and the resulting complex interactions. In this paper, we present a modular trace-based simulation framework, targeting the performance analysis of stream-oriented applications on complex MPSoC architectures. Our framework can analyze systems with a large number of hardware components, while considering various aspects like resource sharing, multi-hop communication, and memory allocation. We demonstrate the potential of our framework by real-life case studies and obtain a speedup of several orders of magnitude and an average accuracy of 97% when compared with the execution on a commercial instruction-accurate simulator.**

I. Introduction

In order to handle the ever increasing complexity of modern embedded applications, the architecture design is shifting from single processors towards heterogeneous multi-processor systems-on-chip (MPSoC). While offering high scale integration, high computing power, and low power consumption, these systems are as well characterized by a large design space. Designers have a large degree of freedom for partitioning parallel application tasks, allocating concurrent hardware components, binding application tasks to hardware components, and choosing resource sharing schemes. This is further aggravated by the steady growth of the number of processing components integrated in the system. For instance, the next generation of the Atmel Diopsis chip [1] will integrate multiple tiles interconnected via a network-on-chip (NoC), and each tile is composed of an ARM9 and a VLIW DSP core.

It is well acknowledged that the design of such complex MPSoCs requires performance evaluation and validation techniques during the whole design trajectory. Because of the overall system complexity, fast estimation methods at early design stages are critical for the exploration of large design spaces. Traditional simulation-based techniques, e.g. Cadence's Virtual Component Co-Design (VCC) [2] and CoWare's Virtual Platform Analyzer (VPA) [3], suffer from long run times and a high set-up effort for each new design. The alternative is using formal analysis, e.g. Modular Performance Analysis (MPA) [4] and SymTA/S [5], which enables fast estimations. However, the drawback of formal analysis is that for highly complex applications and underlying architectures it provides over-pessimistic results which leads to over-provisioning or under-utilization of resources.

Therefore, for the performance estimation of complex MPSoCs, a trade-off between speed and accuracy needs to be moderated. On one hand, the accuracy should be comparable to classical cycle- or instruction- accurate simulations and the estimation method should nevertheless consider real-time aspects like resource sharing or memory allocation schemes. On the other hand, the run time should be acceptable, especially when the target system becomes larger and more complex. The primary questions are therefore, how to estimate the performance of these systems and at what abstraction level, such that the best trade-off between speed and accuracy is achieved.

To answer these questions, we propose a system-level simulation driven by the application trace, targeting the design of stream-oriented applications. Our design methodology is based on the orthogonalization of concerns [6], separating applications from architectures. The application functionality is abstracted as *partially ordered traces of events*, independent of any underlying architecture or mapping. Then, architectural elements are modeled as *virtual machines* which, under specific scheduling polices, consume trace events and dispatch them to subsequently connected virtual machines if needed. In this way, our framework can simulate large and complex MPSoCs, while considering concrete

aspects like resource sharing, memory allocation, and multi-hop communication.

As a result of both, abstraction and thorough modeling of essential elements, our trace-based simulation can achieve a surprisingly fast speed and good accuracy compared to a commercial instruction-accurate simulator [3], i.e. several orders of magnitude speedup and average 97% accuracy. We investigate the effects of memory allocation, multi-hop communication, and different scheduling schemes when mapping an MPEG-2 application onto the multi-tile Atmel Diopsis architecture. The scalability of our approach is demonstrated by mapping a (scaled) MPEG-2 onto multi-processor systems, with up to 16 processors.

II. Related Work

Traditionally, simulation-based techniques are widely used for performance estimation of MPSoCs, e.g. [2], [3] at register-transfer level (RTL) and [7], [8] at system-level. The drawback is that the application functionality has to be executed for each run of the simulation, consequently leading to a slow estimation. Therefore, such techniques are not suitable for system-level performance estimation, especially when design space exploration is needed.

Trace-based performance estimation is a common practice, mainly for the evaluation of memory and cache subsystems in embedded systems, e.g. [9], [10]. The idea of abstracting the functionality of the application as execution traces to speed-up the simulation of the whole system is not new as well. An early report of using application traces in system-level performance estimation is presented by Lahiri et al. [11]. This work focuses on exploring on-chip communication architectures, without considering the modeling of applications and resource sharing schemes.

An integer-controlled dataflow trace-based transformation technique is proposed by Pimentel et al. [12] to refine abstract communication primitives in order to match the architecture. This approach targets communication refinements of mixed accesses of local and shared memories via a shared bus. A more recent work proposed by Wild et al. [13] uses trace-based simulation, targeting network processor architectures that consist of computation and memory modules communicating via a shared SoC bus. The drawback of these two last approaches is that the communication takes place only via a shared bus. Modeling of data transmitted over multiple buses is missing, which is becoming important, as more complex communication schemes will be adopted in future MPSoCs.

A major common lack in all above related works is that none of them considers preemption. Recently, a technique to support modeling of preemptive scheduling is proposed in the MESH trace-based framework [14] by means of including a speculative scheduling together with a rollback mechanism (proposed to overcome misspeculations in the case of interrupts) in its customized thread-based simulation kernel. The drawback is that for frequent rollbacks, the simulation performance drops significantly. By using SystemC, our framework models preemption in a more natural manner without any speculation. Besides, we benefit of faster run-times because SystemC considerably outperforms thread-based simulations by avoiding context-switching, as reported in [15]. Another particularity of MESH is that the communication model uses two abstract methods, i.e. "penalty value" and "exponential function". We argue that neither case properly reflects actual behavior, because the throughput is not always functional to the bus bandwidth, as shown in [16]. In our framework the communication is a first class citizen with assigned arbiters which properly schedule the traffic and therefore can reflect contentions.

Compared to related works, our framework is based on a modular scheme for the composition of the target systems, resulting in a high flexibility and scalability. Therefore, complex MPSoCs can be analyzed with complex communication structures, e.g. multi-hop communications, not being limited to a single shared bus. Furthermore, the SystemC [17] simulation kernel enables our framework to simulate both event- and time-triggered scheduling policies as well as preemption mechanisms.

III. Modeling

Our design methodology is based on the well-known Y-chart [6]. Therefore, the application, architecture, and mapping are specified separately. For the modeling of the mapping, we refer to [18]. This section introduces the used application and architecture models.

A. Modeling of Streaming Applications

For the specification of streaming applications, we adopt the Kahn process network (KPN) model of computation [19] that assumes a network of concurrent autonomous *processes* communicating in a point-to-point fashion via FIFO *software channels* with blocking read and non-blocking write semantics. The process network model of computation is well suited to model streaming applications because it directly exposes the available

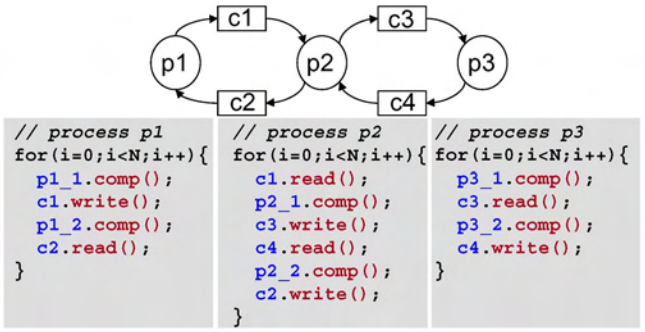

Fig. 1. A simple application and its source code; `x.comp()` represents a computation code segment.

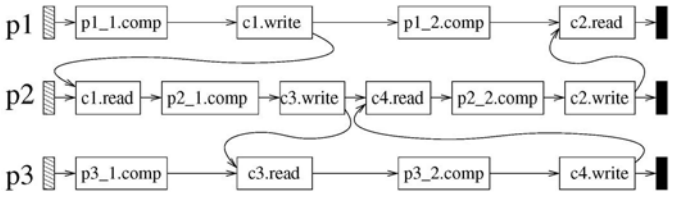

Fig. 2. First iteration of the recurring pattern of $\mathcal{G}$ for Fig. 1. Shadowed and black rectangles denote sources and sinks, respectively.

data and functional parallelism. In addition, the explicit differentiation between computation and communication allows to separately analyze their influence on the target system. Finally, the KPN is determinate, i.e., the result of the computation is independent on the timing. This allows to separate the application functionality from the underlying architecture.

A simple KPN application and the corresponding source code are depicted in Fig. 1. For simplicity, the data rate of each write/read pair is assumed to be constant. We will present later how to tackle dynamically changing data rates for write/read pairs.

1) Trace Definition: The explicit modeling of computation and communication in KPN enables a compact representation for application execution traces. Extracted from the functional simulation, the application execution traces are abstract event sequences consisting of *communication events* and *computation events*. A communication event corresponds to a channel read or write function call in the source code. A computation event of a process clusters all consecutively executed statements between two contiguous communication events. If there are conditional statements, the actual execution branch counts. The execution traces selectively omit unnecessary details for both computation and communication thereof.

The application execution trace can be represented as an acyclic graph $\mathcal{G} = (V, E, E')$, see Fig. 2. Vertices represent computation and communication events and edges represent unrolled timing dependencies that arise due to the sequential nature of the execution or inter-process synchronization. $\mathcal{G}$ is constructed as follows: (1) Every trace event corresponds to a vertex $v \in V$. (2) Each process has a sink and source pair representing the starting and ending events of its execution. (3) Edges in the set E connect every two consecutive events within a process, the direction of which specifies the chronological dependency. In this sense, the trace sequence of a single process is *totally ordered*, i.e. the chronological order of events is fixed. (4) Edges in the set E' connect inter-process communication events, i.e. write/read pairs, corresponding to data transfers via software channels, the direction of which specifies the data dependency of a communication event pair. Overall, the graph is *partially ordered* because the ordering of certain vertices is not specified, e.g. `p1_1.comp` and `p3_1.comp` in Figure 2.

Computation event vertices include runtime annotations of the executed code segment for different possible processors where the process can be mapped. Communication event vertices are annotated with the amount of transmitted data via the corresponding software channel. If the data rates of a write/read pair on a software channel do not match, the write/read events are split based on their greatest common divisor. In this way, write/read pairs will use the same data rate during the simulation.

We draw the following conclusion with respect to application modeling: Given an application specified in a KPN formalism, its functionality can be abstracted as *high-level trace events*, while data dependencies among processes are captured as a *partial order* between trace events. The resulting *acyclic partially ordered graph* is fixed for a given application and *independent* of any target architecture or mapping. The reason is that the application trace graph reflects the property of Kahn process networks [19] of being determinate, which indicates that given the same inputs for an application, the sequence of output value does not change, regardless of the (different) timings of the processes in the network. Therefore the partially ordered trace graph can be reused during a design space exploration loop.

B. Architecture modeling

One of our primary goals is to simulate large scale complex MPSoCs. In order to fulfill this goal, we use a uniform representation for hardware resources, which does not differentiate between computation and communication resources. Instead, the model defines *virtual machines* for timed processing of trace events and *vir-*

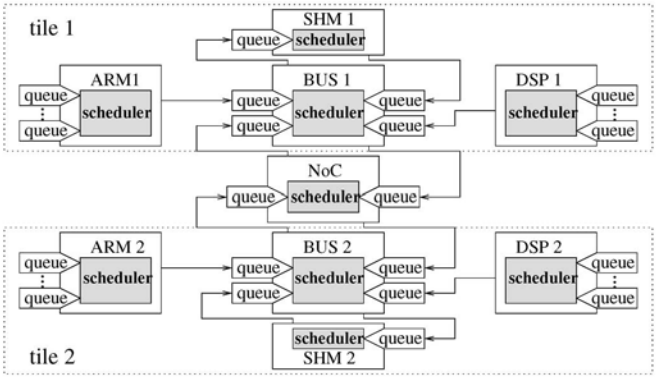

Fig. 3. Architecture model of two-tiles.

tual channels for multi-hop routing of communicated data. Based on this modeling scheme, the architecture is composable and can easily scale to arbitrarily large systems. By simple variations of the interconnects of virtual machines or their scheduling policies, different mappings can be quickly inspected, at an abstract level.

1) Virtual Machine: A virtual machine is defined as an autonomous unit, simulating the timing behavior of a hardware resource. From the representation, we do not differentiate between computation or communication resources. In either case, we model it as a 2-tuple $\mathcal{V} = (S, C)$, where S represents a scheduler and C the capability of the resource, e.g. bus bandwidth or processor frequency. The scheduler manages the mutual access of multiple input event queues. For computation components (e.g. processors), each input event queue corresponds to a process mapped onto it. The scheduler "consumes" trace events from event queues according to its policy and dispatches communication events to connected virtual machines. For communication components (e.g. bus and NoC), each input event queue represents a connected virtual machine. The virtual machine will forward communication events with respect to its arbitration policy and the output communication events are directly dispatched to input queues of connected virtual machines.

As an example, for the shared memory $\mathcal{V}_{SHM}$, S_{SHM} is defined as FIFO, managing the input queue from the connected bus. $C_{SHM} = min(B_{bus}, B_{mem})$ is the minimum between the bandwidths of the memory and the connected bus. In the case when $B_{bus} < B_{mem}$, the throughput of the shared memory is limited by the bus. Otherwise, the bandwidth of the shared memory counts.

Fig. 3 shows the model of a part of our experimental architecture depicted in Fig. 6, where two Atmel Diopsis tiles are connected via a NoC. The local memories of processors are implicit, whereas the shared memory is explicitly modeled as a virtual machine. Each processor has a variable number of queues defined by the processes mapped on it. Each bus holds four different event queues, two for the processors within the same tile, one for the on-tile shared memory, and one for the NoC. The shared memory has only one queue, as it can be accessed only via the shared bus. The NoC has only two queues, corresponding to the two connected buses.

Just by coupling virtual machines through event queues, the system can easily be extended and form any complex architecture. By replacing the scheduler, different scheduling or arbitration policies, e.g. TDMA, round robin, first-in first-out (FIFO), and fixed priority, can be simulated.

2) Virtual Channel: In a complex MPSoC consisting of multiple communication components, the application data may need to traverse multiple hops until they reach the target processor. For instance, if processes `p1` and `p2` in Fig. 1 are mapped to processors `ARM1` and `ARM2`, respectively, a data token transmitted on the channel `c1` will need to traverse `BUS1`, `NoC`, and `BUS2`.

To model this multi-hop communication in a modular way, we define virtual channels, each of which corresponds to a software channel in the application. A virtual channel consists of an ordered set of virtual machines and it is defined according to the sequence of the data transfer:

$$P_{comm}\,(r_1, r_2, ..., r_i, r_{mem}, r_{i+1}, ..., r_{n-1}, r_n),$$

where r_{mem} represents the memory resource where the software channel buffer is actually located and both r_1 and r_n are computation resources. r_{mem} can be located either in the local memory of a processor or in a shared memory.

A virtual channel is explicitly split into a *read path* and a *write path*, interconnected by the same channel buffer r_{mem}:

$$P_{comm}\,(r_1, r_2, ..., r_i, r_{mem}, r_{i+1}, ..., r_{n-1}, r_n) = P_{write}\,(r_1, r_2, ..., r_i, r_{mem}) \cup P_{read}\,(r_{mem}, r_{i+1}, r_{i+2}, ..., r_n)$$

The write path is defined as a sequence of virtual machines: $P_{write}\,(r_1, r_2, ..., r_i, r_{mem})$. The flow for dispatching a write event traverses all the resources from r_1 to r_i until r_{mem}, keeping their order of specification in P_{write}. The performance of a write event on $r_k \in P_{write}$ is modeled by the scheduler of r_k immediately after that the trace event is delivered from r_{k-1} to r_k. The software channel buffer will be updated only after that

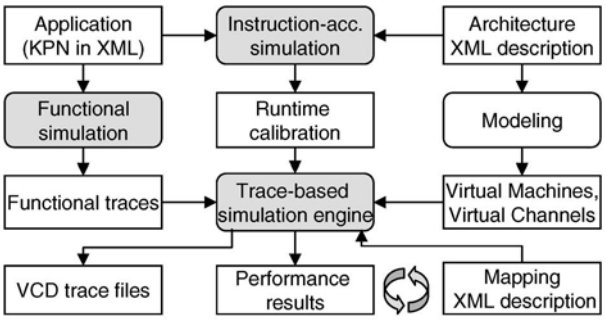

Fig. 4. Overview of the framework.

the resource r_i accomplishes the write to r_{mem}.

The read path is defined in a similar way, as a resource sequence $P_{read}\,(r_{mem}, r_{i+1}, r_{i+2}, ..., r_n)$. Thus, in order to execute a read event mapped on the read path P_{read}, each resource $r_k \in P_{read}$ has to simulate the effect of the read event one after the other, according to their position in P_{read}. In contrast to the moment of updating the software channel buffer in P_{write}, for P_{read}, the first resource, i.e. r_{i+1}, updates the buffer immediately after that it finishes to extract a read event from r_{mem}.

All the read/write scenarios with different locations of the software channel buffer are compliant with this scheme. Using this scheme, the framework can handle complex communications, traversing an arbitrary number of hops.

IV. System Simulation

After introducing the modeling of the application and architecture elements, this section presents how a system is simulated and how its performance is evaluated.

A. Overview of the Framework

Fig. 4 gives an overview of our framework. The three main inputs are the specifications for the application, architecture, and mapping. The application and architecture are specified as previously mentioned, using the process network formalism, e.g. Fig. 1, and using an abstract representation of architectural elements from Fig. 6, respectively. The mapping provides information about the binding of processes to processors and of software channels to write/read paths, as well, about the scheduling policies used on shared resources. The application, architecture, and mapping are all specified in a scalable format, using the Distributed Operation Layer (DOL) XML representation [18] which supports parameterizable specification, therefore enabling the scaling to arbitrarily large target systems.

By means of the DOL framework, the same piece of application source code is used for both the functional simulation, trace generation, and trace calibration. Therefore, the correctness of the application trace and calibration can be safely preserved. The application traces are generated as a result of the functional simulation which itself is automatically generated from the application specification. For generating trace events, the functional simulation is automatically instrumented with the GCC `__LINE__` macro that is used to mark the line number of the start and end points of each code segment. To calibrate the trace events, in the current stage, we use a commercial instruction-accurate simulator, namely CoWare Virtual Platform Analyzer [3]. We apply the same technique, i.e. use the `__LINE__` macro, to track the execution time of each code segment. In order to decouple the event traces from the calibration data and to minimize the memory required to store the event traces, each code segment is annotated with the average execution time for a complete run of the application. In principle, there is no difficulty in annotating each trace event with the exact execution time.

Both the functional simulation and instruction-accurate simulation are executed only once in order to extract execution traces and calibration information, respectively. Afterwards, the trace-based simulation can run completely independently and can be part of a design space exploration loop, where it can be used to inspect different bindings of the application elements to hardware resources, different locations of the communication buffer (i.e. in a local memory or a shared memory), as well as different schedulings.

The main outputs of this trace-based framework are on the one hand performance statistics, such as the execution time of the application, utilization status of both processors and buses, and maximum buffer fill levels for the different channels. On the other hand, we can visualize simulation traces by using the SystemC build-in library, i.e. Value Change Dump (VCD), which allows to inspect dynamic aspects in details, like for instance preemption.

B. Preemptive Scheduling Engine

Modeling preemption in the trace-based simulation at system-level is difficult, since there is no predefined synchronization point and the preemption might take place at any time, even within a trace event. MESH [14], for instance, proposed a complex rollback mechanism to recover the misspeculations due to missed interrupts. Using the SystemC discrete-event simulation kernel, we tackle this problem in a more elegant way. We take advantage of two principles of the SystemC kernel, i.e.

Algorithm 1 Scheduling of A Virtual Machine $\mathcal{V}$

```
1: instantiate SystemC thread
2: while has trace events do
3:     fetch trace event e from G w.r.t. scheduling
4:     if S == TDMA then
5:         wait on associated TDMA slot length
6:     else if S == FIFO then
7:         wait on latency of e
8:     else if S == Fixed Priority then
9:         wait on latency of e and interrupt
10:    end if
11:    update timeslice of e with the actually advanced time
12: end while
```

first, immediate notifications do not advance simulation time and second, the earliest timed event is advanced when no more notifications are available.

Alg. 1 depicts how our framework tackles preemption. Each architectural virtual machine is simulated by a SystemC thread processing "ready" events e from the partially ordered application trace graph $\mathcal{G}$, which are chosen with respect to the scheduling policy S. For instance, for fixed priority, the first event from the highest priority ready queue is chosen, whereas for TDMA, events are chosen depending on associated time slots. The `wait` function returns with the simulation time advanced by the event latency, or will be interrupted if a higher priority process on the same virtual machine becomes available. In the latter case, the simulation time is advanced by the shortest timed event chosen from all virtual machines within the system. After that `wait` returns, the latency of the initial event will be updated with respect to the current simulation time, i.e., subtracting the time advanced by the shortest timed event in another virtual machine.

The advantage of this approach is that the simulation time can advance precisely to the point where preemption occurs, without any speculation. Therefore, the performance of the simulation does not degrade due to the preemption.

V. Experimental Results

To demonstrate the capabilities of the proposed approach, we estimate the performance of an MPEG-2 video decoder application mapped on one or several Atmel Diopsis 940 tiles. Our trace-based simulation is compared with an instruction-accurate simulation implemented using CoWare Virtual Platform Analyzer (VPA) [3].

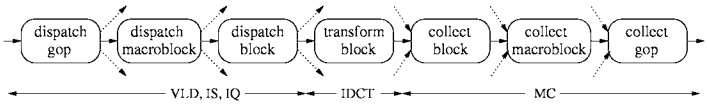

Fig. 5. MPEG-2 decoder process network.

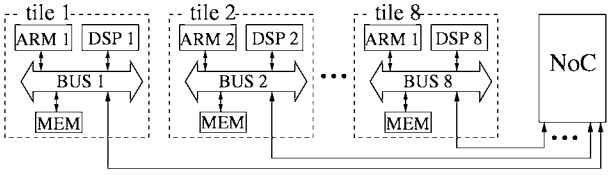

Fig. 6. Block diagram of the target architecture.

A. Case Study Settings

The MPEG-2 decoder is parallelized and implemented as a process network. Fig. 5 shows how we exploit the available data parallelism and functional parallelism in this implementation. To achieve functional parallelism, variable length decoding (VLD), inverse scan (IS), inverse quantization (IQ), and motion compensation (MC) are performed at macroblock level. The inverse discrete cosine transform (IDCT) is performed at block level. These operations can be performed in parallel on subsequent (macro)blocks, leading to the pipeline shown in Fig. 5. To achieve data parallelism, the process network is designed in a scalable manner where compressed data can be dispatched to a parameterizable number of concurrent processes from `dispatch gops`, `dispatch macroblock`, and `dispatch block`, and the decoded data are reassembled in processes `collect block`, `collect macroblock`, and `collect gops`, as indicated by the dotted arrows in Fig. 5.

The considered architecture consists of 8 heterogeneous Atmel Diopsis 940 tiles [1] interconnected by a network on-chip (NoC), as depicted in Fig. 6. Each tile is composed of an in-tile shared memory, a VLIW DSP core and an ARM9 core that both work at a clock frequency of 100 *MHz*. They are interconnected through an in-tile bus that can provide a maximum throughput of 400 *MByte/s*. The NoC has as well a maximum throughput of 400 *MByte/s*. The simulated MPEG-2 clip has a frame-rate of 25 *fps*, bit-rate of 8 *Mbps*, and resolution of 704×576 pixels. For our experiments, we use a video clip with a duration of 15 seconds.

We measure two different timings: (1) *Simulation runtime* (`Sim.`), corresponding to the time elapsed from

the start of the simulation until it completes, as a measurement with the stopwatch. This indicates how fast the execution on a simulation platform is. (2) *Estimated execution time* (`Est.`), indicating how long the application actually needs during the execution on the target architecture, e.g. the time of an MPEG-2 clip running on the Diopsis board. All times are rounded to seconds.

B. Results

First, we compare our trace-based result with a VPA simulation which currently implements a single Diopsis 940 tile with FIFO scheduling policy for all hardware resources. Both the VPA and our trace-based simulation (TSim) execute on a 2 *GHz* AMD Athlon 2800+ Linux machine.

We compare the Est. time and Sim. time for three different mappings of the process network depicted in Fig. 5 onto the Diopsis tile: 1) All processes are mapped to the ARM core and all inter-process communications are done via the local memory; 2) The same process mapping, but all communications are done via the shared memory; 3) The most computation intensive process, i.e. `transform_block`, is wrapped to the DSP core and the inter-processor communications are done via the shared memory.

TABLE I
EST. TIME AND SIM. TIME FOR THE THREE MAPPING CASES, WHEN PROCESSING A 15S VIDEO.

	Est. time (hh:mm:ss)			Sim. time (hh:mm:ss)		
	TSim	VPA	Error	TSim	VPA	Speedup
1	1:36:51	1:38:26	-3%	0:03:30	30:34:00	611
2	1:54:13	1:52:20	+2%	0:03:30	31:30:00	630
3	0:17:17	0:16:46	+3%	0:03:28	74:19:00	1466

Table I indicates the *high accuracy* and *fast performance* of our framework for all three cases. First, we obtain a *very good accuracy of 97%*, due to the thorough modeling of all essential elements in our framework. One of the major reasons of the obtained error is the calibration, especially for computation events where we consider just a global average time. Related to the simulation time, one observation is that VPA simulation time doubles when an additional processor is integrated, i.e. `case 3`. In contrast, the trace-based simulation time is (almost) constant for the three cases. This is due to two main reasons, (a) the application trace does not change as the application remains the same and (b) just single-hop communication is used, which prevented from introducing new trace events (as it is actually the case during the multi-hop communication). Moreover, our approach achieves tremendous speedup, i.e. three orders of magnitude speedup in the case 3, as more architecture resources involved. The reason is that besides the above-mentioned constant traces and single-hop communication, simulated time is advanced by the SystemC kernel and no further synchronization between any architecture resources is needed.

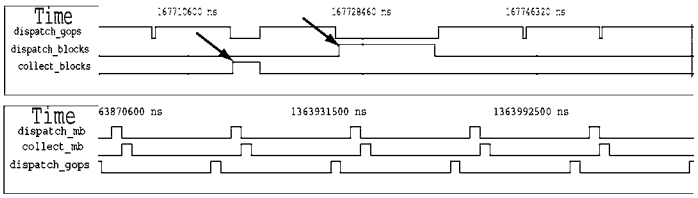

Fig. 7. Trace-based simulation waveforms with FP scheduling (top) and TDMA scheduling (bottom).

Second, we depict the capability of the framework to reflect the impact of the memory allocation. According to Table II, we again obtain in average *97% accuracy*. This comes from the fact that our TSim can model the context switches caused by the different sizes of software channels.

TABLE II
ESTIMATED EXECUTION TIME FOR VARYING CHANNEL SIZES FOR CASE 1.

	Est. runtime (hh:mm:ss)			
Chan. size	600 bytes	1 Kbyes	8 Kbyes	16 Kbyes
VPA	1:56:38	1:44:25	1:38:52	1:38:26
TSim	1:53:03	1:45:35	1:37:24	1:36:51
Error	-3%	+1%	-2%	-2%

The third feature which we investigate is the capability of modeling event- and time-triggered scheduling. Since the VPA currently supports only the FIFO scheduling, we show the resulting waveforms of the trace-based simulation. We reuse the settings of the `case 3` and replace the scheduling policy of the ARM core with both fixed priority with preemption (FP) and TDMA. The corresponding results are in the top (for FP) and bottom (for TDMA) parts of Figure 7. As shown in the top part, the lower priority task `dispatch gops` was preempted by higher priority tasks `collect block` and `dispatch block`, respectively. In the TDMA case, each time slot is 3 μs and the cycle length is 18 μs, which is directly reflected in the figure.

The last feature considered in our experiments is the capability of simulating large systems. In this case, we use a scaled version of the application, where the process `dispatch gop` will dispatch groups of pictures to four parallel routes and each concurrent `dispatch block` will again dispatch data to two `transform block` processes; resulting in totally 26 processes. We applied

different mappings, using up to *8 tiles*. As shown in Table. III, the result is promising. The Sim. time that our framework spends increases only by 18 % even when the system contains 8 times more hardware resources. The difference is mainly due to the fact that multi-hop communications are now present (appearing from the 2-tiles system) and that trace events are generated dynamically to simulate these multi-hop communications.

TABLE III
SIMULATION TIME FOR 1, 2, 4, 8 TILES.

Simulation time (hh:mm:ss)			
1 tile	2 tiles	4 tiles	8 tiles
0:05:08	0:05:29	0:05:43	0:06:4

VI. CONCLUSION

In this paper, we propose a trace-based simulation framework, targeting fast performance analysis for complex MPSoCs. Our framework takes into account a variety of parameters like resource sharing, memory allocation, and data dependencies, having a 97% accuracy when compared with an instruction-accurate simulator. Arbitrarily large MPSoCs can be simulated several orders of magnitude faster compared to an instruction-accurate simulator, where the simulation time only slightly increases with the size of the MPSoC. This implies that our framework is capable of estimating the performance of complex and large MPSoCs with complex communication structures, being in particularly suited for (semi-) automatic design space exploration at system level. The experimental results prove the validity and scalability of the proposed framework.

ACKNOWLEDGEMENTS

This research has been funded by European Integrated Project SHAPES under IST Future Emerging Technologies – Advanced Computing Architecture (ACA). Project number: 26825.

REFERENCES

[1] P. Paolucci, "The Diopsis Multiprocessor Tile of SHAPES," *MPSOC'06, 6th Int. Forum on Application-Specific MPSoC*, 2006.

[2] "The Cadence Virtual Component Co-design (VCC)," `http://www.cadence.com/products/vcc.html`.

[3] "CoWare Virtual Platform Analyzer," `http://www.coware.com`.

[4] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System Architecture Evaluation Using Modular Performance Analysis: A Case Study," *Int'l Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 6, pp. 649–667, Nov. 2006.

[5] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis - The SymTA/S Approach," *Proc. Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, Mar 2005.

[6] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization Of Concerns and Platform-Based Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, Dec 2000.

[7] A. Baghdadi, N.-E. Zergainoh, W. O. Cesário, and A. A. Jerraya, "Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems," *IEEE Trans. on Software Engineering*, vol. 28, no. 9, pp. 822–831, 2002.

[8] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid Design Space Exploration of Heterogeneous Embedded Systems Using Symbolic Search and Multi-Granular Simulation," in *Proc. of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES)*, Jun. 2002, pp. 18–27.

[9] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria, "A Design Framework to Efficiently Explore Energy-Delay Tradeoffs," in *Proc. of the international symposium on Hardware/software codesign (CODES)*, 2001, pp. 260–265.

[10] T. D. Givargis, H. Henkel, and F. Vahid, "Interface and Cache Power Exploration for Core-Based Embedded System Design," in *Proc. of the IEEE/ACM int'l conf. on Computer-Aided Design (ICCAD)*, 1999, p. 270.

[11] K. Lahiri, A. Raghunathan, and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 768–783, Jun. 2001.

[12] A. D. Pimentel, C. Erbas, and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Trans. on Computers*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[13] T. Wild, A. Herkersdorf, and R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures Using Trace-Based transaction fLevel Simulation," in *Proc. Design, Automation and Test in Europe (DATE)*, 2006, pp. 248–253.

[14] F. R. Johnson and J. M. Paul, "Interrupt Modeling for Efficient High-Level Scheduler Design Space Exploration," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 1–22, 2008.

[15] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, "Scalably distributed systemc simulation for embedded applications," Int'l Symp. on Industrial Embedded Systems (SIES), Jun. 2008, pp. 271 – 274.

[16] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini, "A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness," *Int'l Journal of Parallel Programming*, vol. 36, no. 1, pp. 3–36, February 2008.

[17] "Open SystemC Initiative (OSCI)," `http://www.systemc.org`.

[18] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)*, Bratislava, Slovak Republic, Jul. 2007, pp. 29–40.

[19] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress*, North Holland Publishing Co, 1974.