High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations

Roman Plyaskin*, Alejandro Masrur‡, Martin Geier‡, Samarjit Chakraborty‡, Andreas Herkersdorf*

*Institute for Integrated Systems, ‡Institute for Real-Time Computer Systems
Technische Universität München, Arcisstr. 21, 80290 Munich, Germany
roman.plyaskin@tum.de

***Abstract*—Due to the growing complexity of multiprocessor systems-on-chip (MPSoCs), there is an increasing demand on efficient design space exploration techniques. In addition to the analysis of diverse hardware architectures, these techniques should assist the designer in flexible evaluation of various scheduling policies and application mappings while taking effects of the shared on-chip communication infrastructure into account. Most available simulation approaches are either unable to cover all these aspects jointly or have poor simulation performance. In this paper, we present a framework for timing analysis of MPSoC architectures using abstract and yet accurate traces. The traces capture both precise processing latencies and memory access patterns and represent application- and OS-related workload. Performance estimation is performed by an interleaved execution of the traces on a highly configurable multiprocessor platform modeled in our trace-driven SystemC TLM simulator. Using the flexible scheduler model presented in this paper, various mappings and scheduling policies can be rapidly evaluated while considering on-chip interconnect contention and usage of shared resources. Due to the abstraction of the trace-driven simulations, the proposed framework allows for both fast and accurate explorations of MPSoC design alternatives.**

I. Introduction

In modern embedded systems, the amount of functionality merged into single chips is increasing continuously. As a result, system designers have to deal with the constantly growing complexity of MPSoC architectures. The MPSoC paradigm has introduced new challenges for software developers with respect to the efficient use of CPUs, also referred to as processing elements (PEs). Parallelization of existing sequential software often requires non-trivial identification of independent code blocks. In many cases, the software has to be re-written completely in order to utilize multiprocessor architectures. However, in various domains, e.g., multimedia or consumer electronics, the efficient utilization of an MPSoC can be achieved by integrating multiple independent or weakly dependent applications on the same platform (Fig. 1).

If multiple applications need to share PEs, the use of an operating system (OS) becomes essential for resource management. Since porting the target applications to a specific OS is a very time-consuming process, the developer would benefit from a high-level simulation available earlier in the design process. Using the simulation, the designer can perform timing analysis of the whole MPSoC platform, in which diverse scheduling policies, various mappings of the applications to PEs as well as effects of the shared communication can be considered at very early stages.

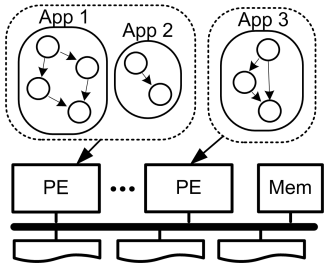

Fig. 1. MPSoC platform executing multiple applications

System-level simulations often have contradicting requirements on accuracy and simulation performance. For timing analysis of complex multiprocessor architectures, cycle accurate CPU models are not feasible in terms of performance due to many low-level details and high complexity of the models. Therefore, for rapid performance evaluations, MPSoC components should be modeled at a higher level of abstraction. As a trade-off between accuracy and performance, trace-based approaches have been proposed [1], [2], [3], [4], [5]. Traces capture the software execution in the abstracted form and, thus, accelerate the exploration of MPSoC architectures.

In this paper, we propose to extend the existing trace-based approaches to support high-level evaluation of scheduling policies and mappings of the target applications while considering fine-grained memory traffic. Our framework is based on abstract and yet accurate traces for both applications and OS-related workload. The traces are obtained using a cycle-accurate CPU simulator and, therefore, contain precise timing latencies and memory access patterns of the applications and OS functions.

In the proposed workflow, the generated traces are reused multiple times during design space exploration. We employ a SystemC TLM trace-driven simulator to perform system-level simulations by the *interleaved execution* of the traces on a multiprocessor hardware platform model. For that purpose, we have created a model of a scheduler which controls the trace execution on the underlying PEs. The scheduler model is highly configurable and allows evaluating various preemptive scheduling policies. This model can be applied for both single- and multiprocessor architectures. In our investigations, we consider general MPSoC architectures consisting of multiple PEs with local caches, shared hierarchical on-chip interconnect and memory components. Since the bus traffic is captured precisely in the traces, the impact of the additional access latencies induced by shared resources and on-chip interconnect can be

studied more accurately, while retaining high-level scheduling of the running applications.

The remainder of this paper is organized as follows. Section II gives a short comparison of our approach with the related work. Section III describes the workflow for MPSoC design space exploration using traces, while section IV deals with the proposed high-level scheduler model. The results of our experiments are summarized in section V. Finally, section VI concludes the paper.

II. Related Work

Consideration of various task mappings and scheduling polices at system level has been addressed in several research works before. A. Gerstlauter et al. introduced a highly abstracted RTOS model implemented in SpecC language [6]. In this framework, the RTOS model provides interface functions that are used by a task model for requesting the RTOS services. R. Le Moigne et al. [7] employed an RTOS model for the analysis of task scheduling at very early design stages. In this method, abstracted functional task models are supplied with additional RTOS function calls for task management. In [8], an RTOS model is used for scheduling and synchronization of applications modeled using task graphs. Each node in the task graph represents an atomic scheduling entity characterized by its execution time. S. Yoo et al. [9] proposed to generate timed OS models using the same methods applied for generation of the final OS code. The above mentioned methods follow a top-down flow of the system design from abstract application models towards the final implementation. In contrast, our approach aims at platform-based MPSoC design for applications for which the executable is already available.

In order to improve accuracy of system-level simulations, instruction set simulators have been used along with an abstract RTOS model [10], [11], [12]. In [12], the authors proposed to co-simulate an ISS and a trace-driven simulator. However, we generate traces before starting the simulations. In this way, the simulation performance is not restrained by the ISS, thus, enabling faster explorations of multiprocessor architectures.

Since an operating system is co-executed with the applications, the timing characteristics of the tasks may get affected due to the additional utilization of the underlying hardware resources by the OS. In many approaches [12], [7], [13], [14], the corresponding overhead is modeled by notating the associated time delays to the OS model. However, on a multiprocessor platform, the CPUs typically compete for a shared resource, e.g., a HW peripheral over the shared arbitrated bus. Therefore, the use of annotated delays is not sufficient for taking these effects into consideration. In many other methods, OS timing characteristics are generally not considered [8], [10]. In the proposed approach, the OS workload is represented in the form of traces containing both communication transactions and fine-granular processing delays. The OS-related traces are then co-executed with the application traces. Thus, the OS effects can be studied more accurately with a marginal loss of simulation performance.

Trace-based approaches have been widely used in high-level modeling of SoC architectures. Spade and Sesame frameworks [4], [5] employ traces to capture computation and communication requirements of target applications modeled using Kahn Processing Networks. By means of a special mapping layer, the trace entries can be mapped and scheduled on various hardware architectures. However, in both frameworks the traces define coarse-grained communication between the KPN processes, and memory accesses imposed by the processes themselves are not considered. Thus, accurate bus contentions cannot be modeled at this level of abstraction. T. Isshiki et. al [3] proposed to use trace-driven workload models that can be mapped and handled in the processors using non-preemptive scheduling. However, in this framework accurate memory access traffic is not considered as well. Mahadevan et al. [8] proposed to employ very accurate traces using an instruction set simulator. However, this approach has a limited support for flexible task mapping and scheduling strategies. In turn, the methodology proposed in this paper allows both for enhanced scheduling capabilities as well as for accurate analysis of the contentions on the shared on-chip interconnect.

III. MPSoC exploration using traces

A. Description of the workflow

The purpose of our framework shown in Fig. 2 is to assist the system designer in the exploration of MPSoC architectures on which multiple applications have to be executed. In the presented workflow, timing analysis of the applications is performed on a highly configurable model of a multiprocessor platform controlled by the scheduler model. Thus, bottlenecks of the underlying architecture can be identified very early, providing the designer with approximated timing estimations.

In the first step, each application is executed on a cycle-accurate simulator of the target CPU. At this point, the stand-alone applications run in the context of a single-core architecture without the notion of an operating system. The information produced by the simulator is further abstracted and used for trace generation. A trace (Fig. 3) is a sequence of pseudo-commands which represent the activity of the CPU captured during the execution of the target application. Particularly, traces contain processing latencies interleaved with communication transactions, e.g., accesses to a memory component. Each latency represents internal processing performed by the micro-architectural components on the register data. The key assumption made at this abstraction level is that the latency values do not depend on whether the CPU is operating as a single unit or as a part of a multiprocessor architecture. The influence of other CPU components occurs during load and store

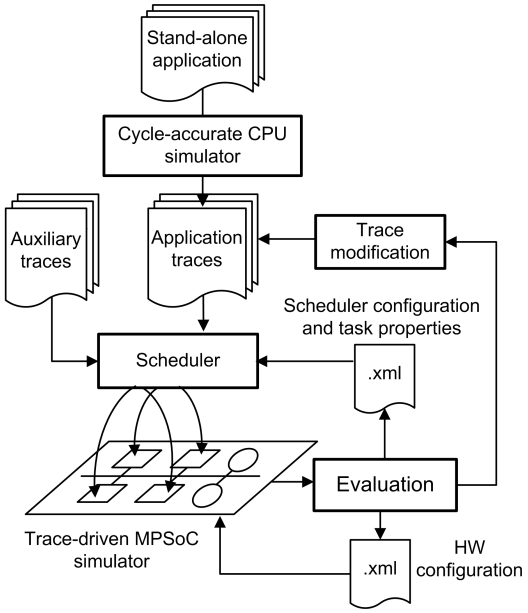

Fig. 2. Workflow for design space exploration of MPSoC architectures

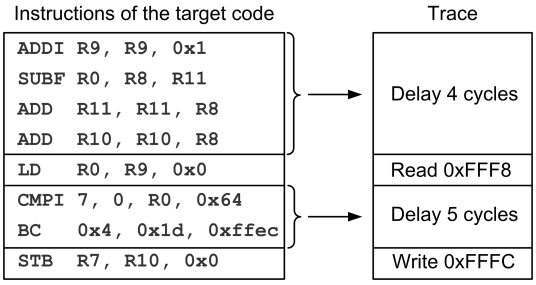

Fig. 3. Representation of an application's execution in the trace form

operations performed via the shared on-chip interconnect. A trace can represent the execution of a complete SW program or a part of the application's code that, according to the designer's expertise, should be later scheduled by the operating system. In addition to the application traces, the designer may include a set of system traces representing additional OS-related workload, e.g., context save/load operations etc.

The resulting traces are used as workload for our trace-driven SystemC TLM simulator. During the simulation, both application and OS-related traces are executed in an interleaved fashion on abstract CPU models controlled by the scheduler model. Scheduling parameters and the configuration of the underlying hardware architecture are specified using XML description files. For example, the designer can set the number of CPUs and the parameters of the on-chip interconnect, specify the memory hierarchy and define how traces should be mapped and scheduled on the CPUs. Depending on the simulation outcomes, the designer can iteratively modify the platform configuration as well as change the mapping and scheduling policies in order to find the optimal architecture which satisfies the performance requirements of the target applications.

B. Trace execution

At simulation runtime, abstract trace-driven CPU models consequently read and execute the pseudo-commands from the trace files. When executing a delay command, the CPU model calls the SystemC wait function simulating the annotated value of the processing latency. On execution of read or write commands, the CPU makes a request to the data cache model providing the notated target address of the transaction. The cache model is highly configurable and supports various read, write and cache line replacement policies as well as basic coherence mechanisms. We explicitly model the data cache in order to consider data miss penalties in the MPSoC scope dynamically at simulation runtime. In case of a cache miss, the CPU performs a blocking TLM transaction on the arbitrated bus to the memory component. If multiple masters share the bus, the transaction might get blocked until the bus arbiter grants access for the CPU. Thus, the overall timing analysis of an MPSoC is performed by a superposition of traces executed by the CPUs. The current level of trace abstraction takes the effects of data caches into account, whereas timing characteristics of instruction caches are assumed to be reflected in the delay primitives. In order to consider the instruction cache effects during the simulation, the traces can be tagged with the additional information about the instruction cache lines used.

In order to support the OS-oriented execution, we have extended the CPU model to consider preemptions and interrupts. Immediate preemptions triggered by the scheduler are performed using *sc_wait(delay, p)* function, where *p* is a SystemC interrupt event notified by the scheduler. If a preemption occurs during the execution of a processing latency, the CPU model stores the remaining delay as well as the current offset of the trace file in a dedicated trace handle. When the CPU resumes executing the preempted trace, the model waits for the remaining delay and continues the execution of the trace from the interrupted position. Thus, multiple traces can be correctly executed on the same instance of the CPU model when using preemptive scheduling.

C. Classification of traces

Application and OS-related traces are treated differently by the scheduler. For the following explanation, we define a *runnable* as an abstract entity which is handled by the scheduler model. Each runnable represents the context in which the execution of a trace is started or stopped in the simulator. Application traces are mapped and scheduled in the context of *application runnables* (AR). In turn, system traces representing the OS-related workload are managed by the scheduler in the context of *system runnables* (SR).

Before the simulation starts, the designer should specify how the traces should be handled by the scheduler. For the application traces, the configuration file specifies a list of *task descriptions* that are used to parameterize the corresponding ARs (Table I). A task description defines which trace file should be executed in the context of the runnable and describes its invocation parameters. Using task descriptions, the designer can force application traces to be executed on a certain CPU instance specified by the `T_CpuID` parameter. Alternatively, the scheduler can perform global scheduling and decide at simulation runtime on which CPU the trace should run.

In the current version, the scheduler model can perform both local and global scheduling using: (i) fixed priorities assigned by the designer or (ii) dynamic priorities based on the earliest-deadline-first algorithm. Other scheduling policies can be implemented in the model using the dedicated APIs. In addition, the designer should specify a period of a trace (if it should be invoked periodically), its initial start time (phase) as well as a number of trace invocations.

At simulation run-time, the scheduler dynamically creates application runnables, which are then scheduled and executed on the CPUs according to the invocation scenario configured by the user. In turn, system runnables are created when the OS-related execution should be simulated, e.g., during context switches or interrupt service routines.

IV. Scheduler model

The scheduler model shown in Fig. 4 has been implemented as a separate SystemC module that interacts with the CPU models. It is a finite state machine that manages the execution and synchronization of ARs and SRs at simulation runtime.

TABLE I
A task description specifying execution parameters of an application trace

Type	Parameter	Description
string	`T_TraceFile`	Trace file
int	`T_CpuID`	CPU the trace must be executed on
sc_time	`T_Period`	Period of trace invocation
sc_time	`T_Phase`	Initial start time
int	`T_nExec`	Number of trace invocations
int	`T_Priority`	Priority value used in case of priority-based scheduling
sc_time	`T_Deadline`	Deadline value used for EDF scheduling

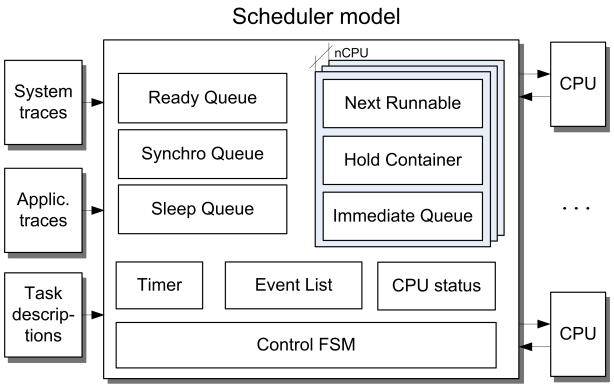

Fig. 4. Structure of the scheduler

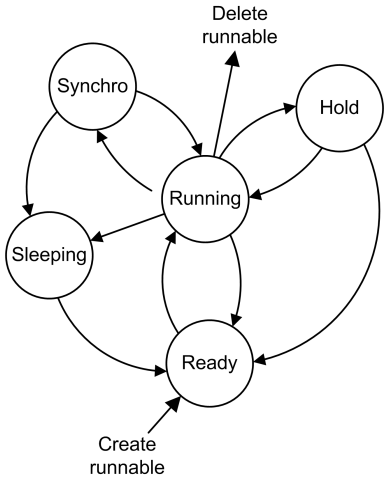

Fig. 5. Model of an application runnable

Application runnables are processed by the scheduler according to the model shown in Fig. 5. Since the ARs are dynamically created at each invocation of the respective application, the model does not have a dormant state, i.e., a state when a task is inactive. Using the invocation parameters from the task description, the internal *timer* of the scheduler calculates time points when ARs have to be created. Afterwards, an AR is placed to the *ready queue* and its initial state is set to `READY`. Additionally, the scheduler decrements the `T_nExec` parameter and calculates the deadline of the runnable (Eq. 1). In case of a periodical trace, the timer is programmed for the time of the next trace invocation calculated using Eq. 2. When the CPU completes executing the trace, the corresponding AR is deleted.

$$Deadline(AR_i) = sc\_time\_stamp() + T\_Deadline; \quad (1)$$

$$Invocation(AR_{i+1}) = sc\_time\_stamp() + T\_Period. \quad (2)$$

The ready queue contains all ARs sorted — depending on whether it is fixed-priority-based or earliest-deadline-first scheduling — by their priorities or deadlines. In our scheduler model, the same queue is used for partitioned and global scheduling. During the partitioned scheduling, a CPU takes only ARs that are mapped to this CPU. Thus, the next running AR is the one which is closest to the queue's top and whose `T_CpuID` corresponds to this CPU. During the global scheduling, the scheduler always selects the first runnable from the queue top and ignores the `T_CpuID` value.

In contrast to application runnables, system runnables are created only when a CPU should simulate a trace representing OS-related workload. For example, the designer can specify traces for the OS scheduler, context switches, interrupt service routines etc. Our framework is capable of modeling external interrupts that force the execution of an ISR system runnable followed by rescheduling of the current ARs.

Being created, SRs are placed into the *immediate queue* of the corresponding CPU. If there is at least one entry in this queue, the scheduler temporarily stops (but not yet preempts) the running AR, places it to the *hold container* and maps the SR to this CPU. After the OS-related trace has finished executing, the AR will either continue executing or will be placed back to the ready queue. We differentiate between these two situations since the execution of a certain OS service may or may not lead to a preemption of the running application. If the running AR is to be preempted (e.g., because another AR with higher priority gets ready), the scheduler will map a context save SR to the underlying CPU and place the preempted AR back to the ready queue. Afterwards, when the preempted AR is about to continue executing, the scheduler will create and map a context load SR respectively.

A. Customization of system runnables

So far system traces have been considered to be fixed for all ARs which might not be true in real systems. For example, task descriptors are typically stored in different memory locations. Therefore, the use of the same context save/load traces for all application runnables is not fully correct.

To address this problem, we have added the possibility to specify a set of custom system traces. The execution of a trace from this subset may be conditioned during the simulation. For example, instead of a common context save SR, an array of SRs can be specified individually for each AR. However, in this particular case the only difference between the context save/load traces will be an address offset in the load and store commands.

Another important aspect which has to be considered is the dependency of the execution time of some OS functions on a current state of the scheduler. For example, the execution of the OS scheduler itself may depend on the number of ARs currently located in the ready queue. Therefore, in this particular case a set of OS scheduler traces can be specified which will be dynamically selected depending on the current size of the ready queue in the model. Thus, accuracy of the timing estimation can be improved compared to permanent SR traces.

B. Synchronization

1) Synchronization mechanisms: An application trace represents the execution of an application without the notion of an OS or other applications. However, in a new MPSoC constellation the designer may want to serialize the trace execution. For example, an application trace may need to be stopped until another trace reaches a certain synchronization point or when a shared resource being used by another AR becomes available. In order to support producer-consumer and mutual exclusive synchronization of traces, our scheduler model provides semaphores and signals triggered by the corresponding trace commands (Table II). These synchronization commands can be easily added to the original application traces. Currently, this is a manual process which, however, can be automated based on the application analysis.

TABLE II
COMMANDS FOR INTER-TRACE SYNCHRONIZATION

Command	Parameter	Description
INIT_SEM	sem_id, value	Initialize semaphore to value
DEC_SEM	sem_id	Decrement semaphore
INC_SEM	sem_id	Increment semaphore
SIG_WAIT	signal_id	Wait for a signal
SIG_FIRE	signal_id	Notify a signal

Semaphores and signals are managed by the scheduler using the *event list* (Fig. 4). It is a database which contains status of the synchronization events. If the required semaphore is negative on decrementing or when an AR has to wait for a signal, the scheduler puts the AR to SLEEPING state after mapping a context save SR.

The *sleep queue* (Fig. 4) is an associative storage for all ARs in the SLEEPING state and the synchronization events they are waiting for. The scheduler processes the queue depending on the type of an arriving event. For example, if a semaphore has been incremented, the scheduler selects one application runnable which is allowed to decrement it according to the rule specified by the designer. In contrast, on a signal notification all of the respective ARs are allowed to continue executing. In both cases, the scheduler moves the selected AR(s) from the sleep queue to the ready queue and generates respective context load SRs.

2) Synchronization balancing: Abstraction of the scheduler allows for flexible evaluation of synchronization mechanisms in the simulator. For example, synchronization can be carried out with or without the OS services by performing, e.g., a busy-wait loop on a shared variable. On the one side, this method can eliminate unnecessary context switchings and reduce the OS overhead. On the other side, the loop-based synchronization may take too long, and the CPU resource will be wasted when there are other ARs ready to be executed.

In order to identify whether the OS-based and OS-free synchronization suits the most for the current application, the runnable model has been extended with additional SYNCHRO state (Fig. 5). In this state, the AR continues running on the CPU. However, instead of executing the application trace, the CPU runs a pre-defined pattern. In most cases, the CPU would be polling a shared variable in the memory; however, the designer is free to create its own pattern. If the required event occurs while the AR is in SYNCHRO state, the CPU will continue executing the application trace and the AR state will be set back to RUNNING.

The user can set a maximum time interval for the SYNCHRO state which is similar to a watch dog timer interval. If the required event has not been occurred within this interval, the AR will be preempted and moved to the sleeping queue. Thus, the designer can easily evaluate possible synchronization options without modifying the application trace.

V. EXPERIMENTAL RESULTS

In this section we demonstrate the proposed workflow for timing estimations of multimedia applications on various MPSoC platforms. For the following experiments, we selected a set of three applications: two instances of a low-priority application performing JPEG image encoding [15] and one high-priority application performing ADPCM encoding of 10 kByte audio samples [16]. The experiments were conducted on a PC with a 2 GHz Intel Core 2 Duo processor and 2 GB of RAM.

The respective traces of the applications were obtained by executing the SW code on a cycle accurate model of PPC e200z6 processor provided by VaST CoMET tool [17]. Table III shows the accuracy of the trace abstraction and the associated increase of the simulation performance. The error in the trace-driven simulation originates from the blocking memory access mechanisms. Currently, possible pipeline stalls or out-of-order execution on cache misses are not considered during the trace generation [2], which is a limitation that we plan to overcome in our future work. In addition to the target applications, we executed the code of the μC-OS/II real-time kernel [18] in order to derive a set of representative traces for the OS scheduler and context switches.

The performance estimation of MPSoC platforms was accomplished in our SystemC-based trace-driven TLM simulator. In order to expand the space of possible design solutions, we identified the most performance demanding part of the JPEG algorithm which was the calculation of discrete cosine transform (DCT). Therefore, in addition to the CPUs, we considered an abstracted model of a DCT hardware accelerator which offloaded the DCT computation from the CPUs. To enable the usage of the accelerator, we replaced the respective parts of the JPEG traces with new trace commands for accessing the component. Moreover, we extended the traces with semaphore primitives for providing mutually exclusive access to the peripheral. On a CPU request, the peripheral model simulated the annotated processing latency of the DCT algorithm and blocked the execution of the application trace until the processing was completed.

During the explorations in the trace-driven simulator, we took four platform architectures (Fig. 6) consisting of CPUs, hierarchical buses as well as local shared memory and an interrupt controller. The architectures differentiated in the number of CPUs and application mappings as follows:

- A_1: CPU_0(adpcm, $jpeg_1$, $jpeg_2$);
- A_2: CPU_0(adpcm, $jpeg_1$, $jpeg_2$), DCT-HW;
- A_3: CPU_0(adpcm, $jpeg_1$), CPU_1($jpeg_2$);
- A_4: CPU_0(adpcm, $jpeg_1$), CPU_1($jpeg_2$), DCT-HW.

For all architectures, the JPEG application processed 5 image frames in the time interval of 300 ms, whereas ADPCM processed 30 audio samples on an external interrupt triggered every 50 ms. The synchronization between the interrupt service routine and ADPCM was carried out using a signal.

In architecture A_1, all applications were mapped to CPU_0 (Fig. 7). Thus, processing of the image frames and the audio samples on this platform resulted in relatively high utilization of 95.8% for CPU_0 and average frame processing time of 242.8 ms for $JPEG_1$ and 287.4 ms for $JPEG_2$. In architectures A_2–A_4, the use of second processor CPU_1 and the hardware accelerator for the DCT computation resulted in a reduced utilization of the CPUs as well as faster execution times of both JPEG applications, as can be quantitatively observed in Table IV. Please note that due to the high abstraction of the traces, the simulation time of the experiments is of the same order of magnitude as the simulated time, which enables fast exploration of the design solutions.

Further extension of the MPSoC platform with additional

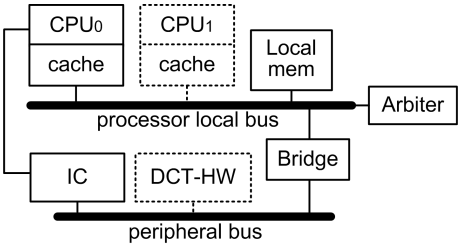

Fig. 6. Platform architectures used in the trace-based simulations: A_1 (1 CPU, without DCT-HW), A_2 (1 CPU, with DCT-HW), A_3 (2 CPUs, without DCT-HW), A_4 (2 CPUs, with DCT-HW)

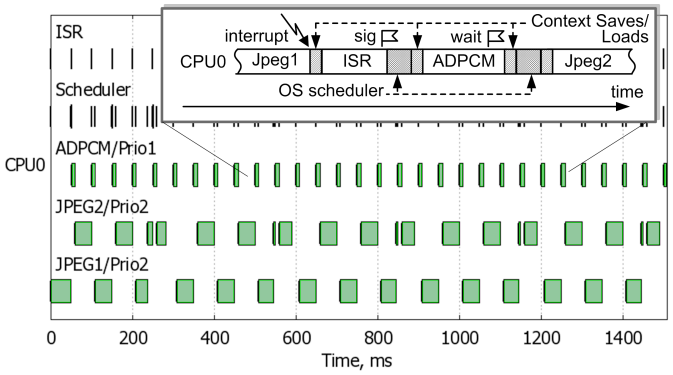

Fig. 7. Execution chart of $JPEG_1$, $JPEG_2$ and ADPCM applications running on CPU_0 without DCT-HW support (platform A1)

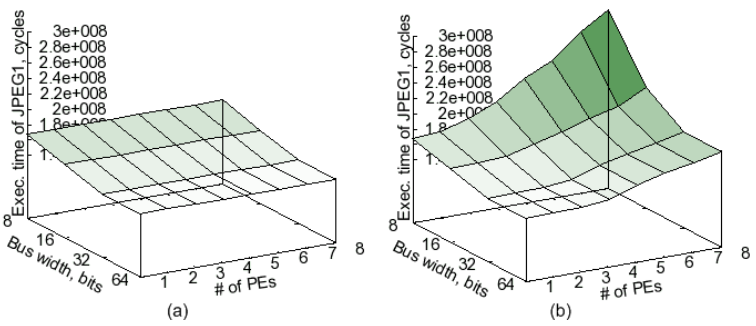

Fig. 8. Average execution time of $JPEG_1$ running on multiple CPUs without DCT-HW in an MPSoC model (a) with fixed memory latencies w/o bus arbitration, and (b) with the consideration of first-come first-served bus arbitration

TABLE III
COMPARISON OF CYCLE-ACCURATE AND TRACE-BASED SIMULATIONS

Appl.	# of inst.	Estimated exec. time, cycles			Simulation time, s	
		VaST	Trace	Error	VaST	Trace
$JPEG_1$	7.1M	11.38M	11.97M	5.1%	0.804	0.587
$JPEG_2$	7.1M	11.39M	11.98M	5.1%	0.886	0.520
ADPCM	0.48M	0.81M	0.83M	1.5%	0.115	0.029

TABLE IV
RESULTS OF TIMING ESTIMATIONS IN THE TRACE SIMULATOR

Arch.	Utilization		Average execution time, ms			Simted time, s	Simtion time, s
	CPU_0	CPU_1	$JPEG_1$	$JPEG_2$	ADPCM		
A_1	95.8%	—	242.8	287.4	50.3	1.51	8.29
A_2	60.5%	—	131.3	162.8	50.3	1.51	6.50
A_3	56.3%	39.9%	143.5	120.3	50.3	1.51	8.97
A_4	38.8%	22.4%	82.5	67.5	50.3	1.51	8.46

CPUs emphasizes the importance of accurate modeling of memory accesses in the trace-based modeling. Fig. 8a illustrates simulation results of the platform with a varying number of CPUs (each executing the JPEG algorithm) without precise modeling of CPU–memory communication. We managed to reconstruct this scenario in our framework by employing a shared bus without arbitration. As the number of masters that can simultaneously communicate over the bus is not bounded, the execution times of the applications running on the MPSoC become underestimated. In contrast, Fig. 8b shows the simulation results of the MPSoC with enabled first-come first-served bus arbitration. Due to the larger bus contention periods, the performance of the JPEG application decreased drastically, which can be clearly observed in the architectures with a low-bandwidth bus (slow-down by factor of 1.71 for an 8-bit bus with 8 CPUs).

VI. CONCLUSIONS

In this paper, we presented a new approach for high-level timing analysis of MPSoCs using abstract and yet accurate traces that represent execution of the target applications and OS-related workload. The execution of traces is managed by the scheduler model which enables fast and flexible exploration of application mappings and scheduling policies. Moreover, the exact memory traffic of the applications captured in the traces allows for examination of shared bus contentions and their implication on the performance of the applications. Currently, our approach aims at independent applications or weakly dependent applications. However, in our future work we will focus on more detailed modeling of inter-process communication using traces. In addition, we plan to increase accuracy of the trace-based modeling by the elaborate consideration of microarchitectural effects during the trace generation.

REFERENCES

[1] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES – trace-based architecture performance evaluation with SystemC," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005.

[2] R. Plyaskin and A. Herkersdorf, "A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces," in *Architecture of Computing Systems - ARCS 2010*. Springer, 2010, pp. 199–210.

[3] T. Isshiki, D. Li, H. Kunieda, T. Isomura, and K. Satou, "Trace-driven workload simulation method for multiprocessor System-On-Chips," in *Proceedings of the 46th Annual Design Automation Conference*. San Francisco, California: ACM, 2009, pp. 232–237.

[4] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with spade: An M-JPEG case study," in *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, 2001, pp. 31–38.

[5] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *Computers, IEEE Transactions on*, vol. 55, no. 2, pp. 99–112, 2006.

[6] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS modeling for system level design," in *DATE, 2003*, 2003, pp. 130–135.

[7] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," in *DATE '04*. Washington, DC, USA: IEEE Computer Society, 2004, p. 30082.

[8] S. Mahadevan, K. Virk, and J. Madsen, "ARTS: A SystemC-based framework for modelling multiprocessor systems-on-chip," *Design Automation of Embedded Systems*, 2006.

[9] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design," in *DATE '02*. Washington, DC, USA: IEEE Computer Society, 2002, p. 620.

[10] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel, "Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation," in *CODES/ISSS '08*. New York, NY, USA: ACM, 2008, pp. 143–148.

[11] J. Chevalier, M. Rondonneau, O. Benny, G. Bois, E. M. Aboulhamid, and F. Boyer, "Space: A hardware/software SystemC modeling platform including an RTOS." in *FDL*. ECSI, 2003, pp. 704–716.

[12] D. Kim, Y. Yi, and S. Ha, "Trace-driven HW/SW cosimulation using virtual synchronization technique," in *DAC '05*. New York, NY, USA: ACM, 2005, pp. 345–348.

[13] H. M. AbdElSalam, S. Kobayashi, K. Sakanushi, Y. Takeuchi, and M. Imai, "Towards a higher level of abstraction in hardware/software cosimulation," in *ICDCSW '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 824–830.

[14] Z. He, A. Mok, and C. Peng, "Timed RTOS modeling for embedded system design," in *RTAS '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 448–457.

[15] Embedded JPEG Codec Library. [Online]. Available: http://sourceforge.net/projects/mb-jpeg/

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Workload Characterization, Annual IEEE International Workshop*, vol. 0, pp. 3–14, 2001.

[17] VaST Systems Technology. [Online]. Available: http://www.vastsystems.com

[18] J. J. Labrosse, *Microc/OS-II*. R & D Books, 1998.