Des Autom Embed Syst (2006) 10:157–179
DOI 10.1007/s10617-006-9589-4

TAPES—Trace-based architecture performance evaluation with SystemC

Thomas Wild · Andreas Herkersdorf · Gyoo-Yeong Lee

Received: 6 February 2006 / Revised: 20 April 2006 / Accepted: 15 May 2006

Abstract The design of today's System-on-Chip (SoC) architectures faces many challenges in respect to the involved complexity and heterogeneity. An early and systematic exploration of alternatives is mandatory to find a solution that meets all design requirements. Therefore, the experience of system architects has to be supplemented with efficient performance evaluation methods and tools that help in the broad exploration of the solution space. This article describes TAPES (Trace-based Architecture Performance Evaluation with SystemC), an approach that supports system designers in the performance evaluation of SoC architectures. The concept captures the functionality of the architecture in the form of traces for each resource. The trace primitives making up a trace are translated at simulation run-time into transactions and superposed on the system architecture. The method uses SystemC as modeling language, requires low modeling effort and yet provides accurate results within reasonable turnaround times. A concluding application example for the exploration of a network processor architecture demonstrates the effectiveness of the TAPES approach.

Keywords SystemC · Performance evaluation · Architecture exploration · Trace-driven simulation · Transaction level modeling

1. Introduction

In the design of System-on-Chip (SoC) solutions the definition and quantitative characterization of suitable architectures is a vital issue. Typical SoCs may consist of a broad range of

T. Wild (✉) · A. Herkersdorf
Institute for Integrated Systems, Technical University of Munich, D-80290 München, Germany
e-mail: thomas.wild@tum.de

A. Herkersdorf
e-mail: a.herkersdorf@ei.tum.de

G.-Y. Lee
Department of Information & Communication Eng., Kangwon National University, Samcheok, Gangwondo 245-711, South Korea
e-mail: gyoolee@samcheok.ac.kr

IP modules like embedded processors, accelerator blocks, interface modules, a memory subsystem and specifically designed HW modules. All architectural resources are connected via a communication infrastructure that provides system-internal connectivity for the exchange of data and synchronization. Choosing an appropriate system architecture and mapping the application's subtasks to the resources are important tasks in the process of system level design [1].

As the design space opens up a vast variety of solution alternatives, a systematic exploration of design alternatives is necessary. Good design decisions in early design phases make up a strong basis for the final implementation steps, both for the hardware and software parts of the system. The designers experience and intuition that guide through the exploration process has to be supplemented with efficient methods and tools for evaluating potential solutions. The insights gained from a thorough analysis can then iteratively be used for directed modifications of architectures. This helps in meeting time-to-market requirements as well as design goals concerning performance, area and power, and leads to the selection of solutions that are traded off against alternatives more intensively, eventually enabling higher design quality.

Starting with modeling and simulating of such a system on RT-level using a hardware description language is not feasible because of the modeling effort, the simulation times and the inability to capture the behavior of mixed HW/SW systems. Therefore, the abstraction level has to be raised. Recently, new modeling languages have been developed in order to support designers in the early design stages on system level. Among others like SpecC or System Verilog, mainly SystemC [6, 18] has gained attraction for design exploration. SystemC allows modeling of SoCs on a high abstraction level and gradual refinement for design and verification purposes.

The definition of the system architecture, i.e. the allocation of architectural resources and the mapping of tasks under given optimization criteria, is the major step in system level design. As exploration is an iterative process a great number of different potential solutions have to be evaluated regarding their compliance with the design requirements. This means that the complexity to generate suitable models for performance analysis as well as the effort for evaluation have to be strictly limited, in order to allow the comparison of as much alternatives as required. On the other hand it is necessary to capture enough information with high accuracy for making reasonable design decisions.

For performance analysis a model is required that captures both the function and the characteristics of the architecture resources adequately. This means that a performance model should record the application's execution behavior on the system architecture and provide means to measure the load values of the computation resources, the communication infrastructure and the memory subsystem under workload conditions resulting from external stimuli. For this purpose it is not necessary to execute the complete functional specification at simulation run-time, but to reproduce the situation as if the application is actually executed on the system architecture. Thus, performance models are usually different from models for functional verification and synthesis, which have to be complete and unambiguous.

In this paper we present TAPES (Trace-based Architecture Performance Evaluation with SystemC), a transaction level approach for the performance evaluation of SoC architectures. Traces are used to model the behaviors of the application on a high abstraction level and their interaction on the architecture, avoiding maintenance of a fully-fledged functional model during architecture exploration. This concept provides high simulation efficiency combined with easy reconfigurability of the underlying model to different resource and mapping configurations.

The paper is structured as follows: In Section 2 we give a short overview of related work in this field. The subsequent two sections describe the concept and implementation of TAPES respectively. Section 5 shows a network processor application that is explored with the help of our approach. Section 6 concludes the article and gives some outlook for the further improvement of TAPES.

2. Related work

Performance estimation on system level is a topic of intensive research. Many approaches have been proposed that rely on different concepts. A Network Calculus based approach is described in [19] that uses performance networks for modeling the interplay of processes on the system architecture. Event streams covering the workload and resource streams describing the service of resources interact in performance components that are connected to a network. The resulting transformations enable the derivation of performance data like resource load values or end-to-end latencies. SymTA/S [7] uses formal scheduling analysis techniques and symbolic simulation for performance and timing analysis. One problem of exact methods is their limited ability to capture real workload scenarios, which may have characteristics that do not fully fit to the parameters of formal approaches. This problem is less critical in simulation based methodologies where real application stimuli are easier applicable. Therefore, in many cases timed simulation is used for performance evaluation.

Transaction level models (TLM) have gained wide acceptance in the system level design community [4]. Decoupling computation from communication and defining interfaces that provide specific functions for modeling abstract communication enable stepwise refinement of TLMs. Nevertheless, TLMs are applied on different abstraction levels and for very different purposes [5]. The major abstraction level relevant for architecture exploration is the level of concurrent processes. However, in both variants, without and with timing information, the abstraction is too high to capture the influence of the communication on system performance. In order to meet the goals of fast evaluation and high precision we concentrate in the following on models that are very abstract in respect to functionality and precise concerning architecture.

Ptolemy [3] is a design framework that targets at modeling, simulation and design of embedded systems with special consideration of different models of computation, however, with the main focus on specification and code generation. The POLIS system [1] supports the designer in modeling and verification of applications represented as CFSMs and guides towards implementation. The commercial tool VCC was based on ideas of POLIS and included the support of multiprocessor systems, however with restricted support of application domains. Metropolis [1] is a design environment for all phases of the design process from concept to implementation. It addresses also performance evaluation through simulation and formal methods using a meta-model that can represent different design aspects like function and architecture models as well as their mapping.

SystemQ [17] applies transaction level modeling in SystemC and uses queuing networks to cover the behavior of system-level platforms. Click [9] is an approach for specifying packet processing functionalities in a very efficient way, however, without providing means for the evaluation of their performance on specific system architectures. StepNP [8] is a network processor evaluation platform that utilizes Click as input specification. The performance simulation part of StepNP uses a SystemC TLM, however, includes full functional models that are executed on ISSs.

Trace driven simulation techniques have widely been used in the performance evaluation of computer systems in general [16] or in the area of multiprocessor systems [14]. In [8, 11] traces recorded from the functional level are mapped to the architecture and are used in the refinement process of the architecture, especially of the communication infrastructure. These approaches mainly rely on traces related to a given architecture and use them in the refinement of the system.

In our performance evaluation approach we use traces in a more general way for the specification of the functionality including the mapping of its subtasks to the architectural resources and for the description of SoC workloads. Traces as used in our concept can be considered as a programming model that provides the flexibility and the fast adaptability of the underlying performance model to architecture modifications. The accuracy of the trace specification directly determines the precision of the simulation results.

3. The TAPES approach

The main issue for performance evaluation is to gain reliable data of the resource usage and processing latencies in order to identify bottlenecks in the system architecture. The system architect, in turn, can use these results for the iterative modification of the system architecture to meet the design requirements. Figure 1 shows our view of the high level design flow with performance evaluation as a part of the architecture exploration loop. Starting from a specification or a fully functional model (like a C program) and taking into account an initial architecture a performance model is built that makes up the input for the performance evaluation step. Depending on the results of the analysis the architecture is iteratively modified until the design requirements are met. The result of this architecture exploration loop is the specification of an architecture that is used in the subsequent implementation phase. In parallel to these steps, functional validation is performed to guarantee compliance to the specification.

As performance evaluation is part of the exploration loop the following three main items have to be considered in respect to both functionality and efficiency of the approach:

- In order to allow for the analysis of a high number of alternative system architectures the effort for generating and simulating the model and for evaluating the results should not be prohibitive for interactive exploration.
- The underlying model has to be adaptable in respect to the number and type of resources that are allocated in the system architecture.

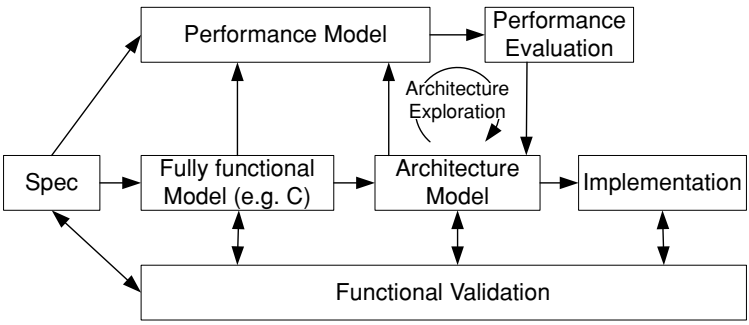

Fig. 1 High level design flow

- The mapping of the tasks to the resources has to be flexible in order to support easy investigation of alternative mapping decisions.

3.1. Basic principles

TAPES addresses these requirements by using a system model that precisely covers the interaction between resources, however, abstracts the execution of the application's tasks as much as possible. Precisely, as we do not aim at a functional verification of the system we replace the processing of the tasks by their execution latencies on the respective resources. The functionalities of each resource are consequently described as sequences of processing delays interleaved with external transactions. The specification of such a sequence is denoted later on as trace. Architecture resources are thus treated as black boxes whose internal structure and the actual internal processing are disregarded during simulation. This abstraction enables higher simulation speed than an annotated, fully-fledged functional model and yet allows capturing the interplay and the sequence of tasks within the system architecture correctly.

In contrast to the computation resources the communication architecture, especially shared communication resources, cannot be abstracted to the same degree. In order to capture the dynamics of resource conflicts, caused by competing transactions from parallel resources, the mechanisms for contention resolution and arbitration are implemented in the simulation model for the communication resources. Too much abstraction in this case would prevent the extraction of realistic performance values. The simulation of the TAPES architecture model thus captures both the data and control flows within the architecture correctly and is therefore completely sufficient for recording usage data of computation and communication resources, required for performance evaluation and identification of bottlenecks in the system.

A further important property of our concept is to separate the specification of the system's hardware structure from the definition of its functionality. The simulation model is dynamically generated at simulation startup time from a library of abstract resource types like CPUs, memories or accelerators and the communication architecture according to the user requirements without the need to modify the model's source code.

The features of TAPES described above enable high efficiency concerning both the simulation of the model and the modification effort for adapting it to different architecture variants during the exploration phase. In the rest of this chapter we outline these issues and the underlying concepts; in the following section we give more information on their actual implementation.

3.2. Modeling of the structure of the system architecture

The system architecture is build of modules that interact with transactions leaving out details of the later RTL implementation that are not needed in that early stage in the design flow. For modeling SoC architectures that consist of an arbitrary number of modules a modular approach is followed to build the system topology. Currently the approach is targeted at bus-based SoC architectures with a single system bus and point-to-point connections between certain modules, as it is shown in Fig. 2. An arbitrary number of modules like embedded CPUs, HW accelerators or memory blocks can be attached to the system bus. The allowable maximum numbers of masters and slaves for the actual system bus, however, have to be taken into account when configuring the system model. Additionally, further modules that are more application specific and mainly interact via point-to-point links can be added to the system architecture as well.

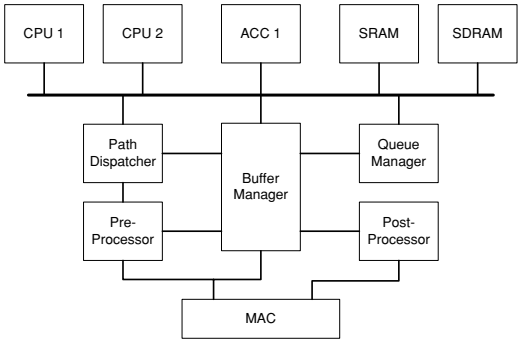

Fig. 2 SoC architecture

Originally, the approach has been developed with network processors as the main focus. Therefore, a couple of specific architecture modules from this application domain is available, like a buffer manager responsible for storing and retrieving variable sized packets in memory, a queue manager and a pre- and post-processor in the ingress and egress data paths. Figure 2 shows an exemplary, on a standard SoC platform based architecture for a network processor implementing our FlexPath concept [12]. However, the principles used in TAPES are generic, so a generalization of the concept is easily feasible.

In order to enable fast changes of the system architecture in the course of the exploration process, at simulation start-up the hardware structure of the simulation model is dynamically built up according to the number and type of resources as specified in the system configuration file. I.e. the user does not have to manually adapt the SystemC description to build a system model for simulation. When interconnecting the resources of the SystemC architecture model the corresponding interface types either bus master, bus slave or point-to-point communication are taken into account. The system configuration file is also used to adjust other architecture parameters, as will be shown in Section 4.

For the definition of the architecture functionality, the user has to provide functional specifications of all modules used in the architecture. This is done with traces that determine the behaviors of the resources in respect to internal processing as well as the communication pattern that can be observed externally, as described in the sequel.

3.3. Modeling of the functionality of the system architecture

The simulation model captures the system functionality by specifying traces for all architecture resources. Each trace represents the sequence of tasks and transactions with other resources that have to be executed in that resource to model the processing of a specific data item. Depending on the partitioning decision, each resource contains one or more traces that are related to different types of processing sequences. Figure 3 shows an example for the processing of an IP packet in an embedded CPU of a network processor.

The processing starts with two read operations, one for getting the reference of the packet to be processed, the other for loading the first bytes of the packet containing its header. After a

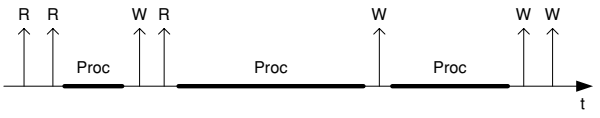

Fig. 3 Trace example

certain processing time—e.g. for checking the validity of the packet—a lookup is performed, represented by a write to a coprocessor and a read to return the result. Then packet processing is continued with an intermediate write operation and finally the modified parts of the packet are written back to the memory and the packet reference is sent e.g. to the queue manager.

In general, a trace in TAPES is a sequence of trace primitives representing either the internal processing of the associated resource or an interaction with other architecture resources, which corresponds directly to a call of a SystemC transaction. In the target resource of the architecture, the call of a transaction in turn triggers the execution of a specific local trace. In addition to these basic trace primitives there are also macros that are translated at simulation run-time into a sequence of basic trace primitives. This trace driven interaction of architecture resources thus enables the simulation of the system behavior.

As the transactions of all architecture resources working in parallel are superposed on shared resources like bus or memory, additional delays are generated as consequence of competing accesses. This effect has also to be reproduced by the simulation model. Therefore, the models for shared resources implement the corresponding mechanisms for the resolution of conflicting accesses as their real world counterparts. In respect to our trace-driven simulation, this leads to a stretching of the traces, as a consequence of arbitration latencies as well as processing time in the called modules. Figure 4 depicts this for the beginning of the trace shown in Fig. 3.

Figure 4 also shows the blocking interaction of a CPU with an accelerator. In a write transaction the required data is first written to the accelerator followed directly by the read of the result. The write operation, when finished, triggers the execution of the corresponding accelerator trace; the subsequent read transaction blocks the CPU until the accelerator has finished and the result is transferred back to the CPU. Thus, trace execution in called modules prolongs trace execution of the CPU as well. Note that our simulation approach also supports non-blocking accelerator calls.

Another consequence of the functional abstraction is that no real data is exchanged during simulation. Only tokens representing a reference to the data currently being processed are transferred in the system model. For network processor architectures, the current main application domain of TAPES, this is a packet. In alternative applications it might be a video frame or more generally an external request. For simplicity, we leave out these alternative types of data in the following description and use "packet" as placeholder for the data to be processed.

The system functionality and its execution can formally be specified in the following way. The behavior of a resource R_i is determined by the set of traces F_i. Each trace $T_{j,i}$ is numbered and represents a specific sub-functionality that is implemented on resource R_i. A trace T_j is an ordered list or vector of trace primitives $p_{l,j}$ from the trace primitive set P that

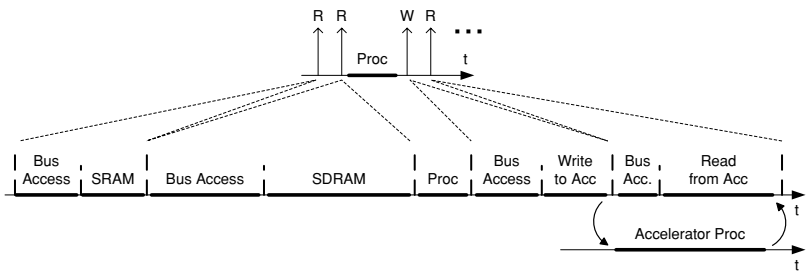

Fig. 4 Transformation of the CPU trace during simulation

defines the elementary operations available in the simulation approach. P is the set union of different primitive sets representing internal processing (P_p), communication with other modules (P_c) or denoting macros (P_m).

$$F_i = \{T_{1i}, T_{2i}, \ldots, T_{ni}\} \tag{1}$$

$$T_j = (p_{1,j}, p_{2,j}, \ldots, p_{k,j}), \quad p_{l,j} \in P \tag{2}$$

$$P = P_p \cup P_c \cup P_m \tag{3}$$

At simulation run-time, processing primitives $p_p \in P_p$ are mapped to the execution of the *wait()* function with a specific delay value. Communication primitives $p_c \in P_c$ in turn lead to the call of an interface function *f()* of the communication channel. Depending on the model of this channel, its load situation and the availability of the slave resource (R_s) of the transaction, this will lead—with a certain delay—to the call of the interface function *g()* of the slave interface of R_s. Eventually, *g()* triggers the execution of trace $T_{h,s}$ in R_s.

$$p_p \rightarrow wait(Delay) \tag{4}$$

$$p_c \rightarrow IF_c :: f(\,);\ IF_c :: f(\,) \rightarrow IF_s :: g(\,);\ IF_s :: g(\,) \rightarrow T_{h,s} \tag{5}$$

The definition of the traces and their execution as described hitherto is static, i.e. all packets experience the identical treatment. Macros $p_m \in P_m$ are special trace primitives that are translated by a function $t_m()$ at simulation run-time into a temporary trace T_{tmp} consisting of processing or communication trace primitives that is then executed in the usual way.

$$t_m\,(p_m, ID) \rightarrow T_{tmp} \tag{6}$$

$$T_{tmp} = (p_{1,tmp}, p_{2,tmp}, \ldots, p_{k,tmp}), \quad p_{n,tmp} \in \{P_p \cup P_c\} \tag{7}$$

The parameter *ID* of function $t_m()$ is a reference to the packet token, which enables access of annotated meta data, like the packet size or other specific information, that can be taken into account when T_{tmp} is generated. The interpretation of macros at simulation run-time makes up a special feature that allows capturing an individual treatment of packets.

The traces of the architecture's resources constitute the user interface for the configuration of the model functionality. We will have a closer look onto the list of concrete trace primitives and how they are evaluated during simulation run-time in Section 4.2.

The trace specification for the system architecture is currently a manual process. The two main issues of this task are the identification of contiguous instructions represented in the model by processing primitives and the evaluation of the associated compound processing latencies. This process has to take into account the binding decision that determines the interleaved communication primitives. The timing behaviors of the different resources can either be retrieved from data sheets, e.g. for memory or specific accelerator blocks, or by recording the activity of embedded CPUs with a logic analyzer in combination with a disassembler, allowing a sequencing of the program execution. A simpler starting point for trace specification are packet processing benchmarks like [15] that give typical instruction profiles for specific network processing tasks.

Nevertheless, this approach enables high flexibility concerning modifications of the binding decision. If a sub-function has to be moved during architecture exploration from SW to a specific HW accelerator, in the simulation model the corresponding processing primitive(s)

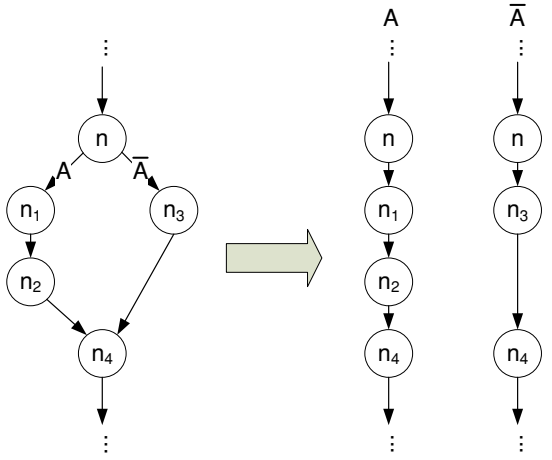

Fig. 5 Resolution of control dependencies

in the CPU traces is/are replaced by communication primitives that capture the transaction with the accelerator module. Further on, the accelerator module has to be registered in the system configuration and characterized by a suitable trace file.

As our approach abstracts processing by its corresponding latencies, control dependencies cannot be resolved at simulation run-time like in the real system. This is particularly important for our main application domain networking, which is heavily control dominated. On the left hand side of Fig. 5 the problem is demonstrated for an application containing a branch that determines further processing depending on the evaluation of a condition A.

In our approach we solve this problem by specifying independent traces for both cases, A being true and false. This is shown in Fig. 5 as transformation of a task graph. In general, for the resolution of an arbitrary number of control dependencies in an application, we evaluate all possible patterns of conditions resulting in graphs without conditional branches. Then specific traces are defined for each combination.

In respect to the work load that stimulates the simulator, however, this procedure means that each incoming packet has to be annotated with the information to which of the cases it belongs. In particular, this identifies the initial trace to be started in the CPU that then determines the complete subsequent processing in the system. For synthetically generated workloads, appropriate stimulation patterns can be generated in an arbitrary way. If real world traffic is available it is preprocessed offline and annotated, so that it can be replayed and used as load for the simulator as often as needed.

4. Implementation

Following a strictly modular approach is the natural consequence of the desire to make the simulation model easily adaptable to different architecture configurations. This applies to the hardware structure that encompasses the architecture resources including the communication architecture as well as to the mapping of the functionality.

4.1. Specification of the system architecture

As described above our approach is based on SystemC TLM, which separates communication from computation. This is done by specifying the interaction between components via

interfaces. An interface is an abstract class that declares methods that are provided by the interface as abstract function calls. The implementation of these methods, however, is defined in the hierarchical channel that is derived from the interface. A module that contains a port corresponding to the respective interface and that is connected to the hierarchical channel can thus interact with it by calling the appropriate interface methods. This allows modeling the communication mechanism, e.g. a bus system with its protocol, independent from the modules that are connected to the bus.

In our evaluation approach we currently support communication architectures with a single common system bus and a specific point-to-point connection network, including also interrupt lines. To implement this communication architecture we have defined the following interfaces types:

- *bus_master_if*: implements read/write access of master modules on the bus

```
class bus_master_if : public sc_interface{
public:
    virtual int read(bus_request &request)=0;
    virtual int write(bus_request &request)=0;
};
```

- *bus_slave_if*: implements arbitration (read_request / write_request) and read / write accesses to the slave

```
class bus_slave_if : public sc_interface{
public:
    virtual int request_read(int source)=0;
    virtual int request_write(int source)=0;
    virtual bool read_transfer(bus_request &request, int
    bus_clock)=0;
    virtual bool write_transfer(bus_request &request, int
    bus_clock)=0;
};
```

The parameter *bus_clock* is used in the bus slaves to calculate response times synchronized to the frequency of the bus.

- *direct_comm_if*: implements point-to-point read / write and interrupt on any resource type (where required)

```
class direct_comm_if : public sc_interface{
public:
    virtual int interrupt(int source, int trace_id)=0;
    virtual int set_indication(int source)=0;
    virtual int remove_indication(int source)=0;
    virtual void write_data(dc_request &request)=0;
    virtual void read_data(dc_request &request)=0;
};
```

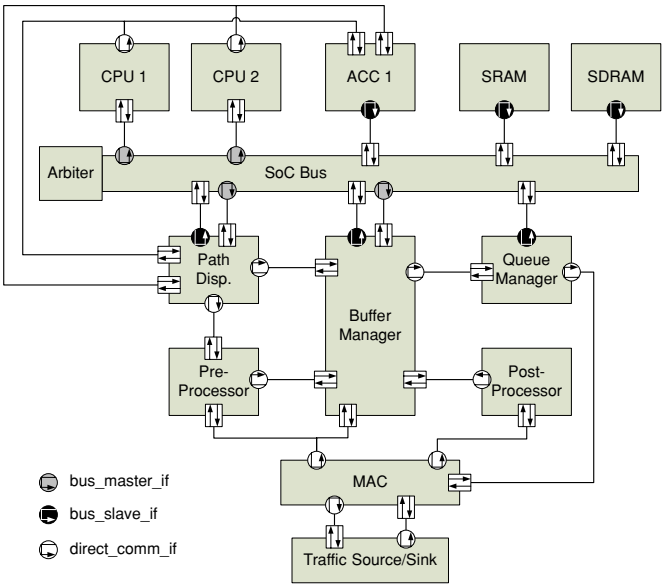

Fig. 6 SystemC model of network processor architecture

Figure 6 shows the model in SystemC notation corresponding to the architecture from Fig. 2 and identifies the type of each interface according to the list given above. The module types shown above the SoC bus, CPUs, accelerators and memories, are either pure bus masters or slaves whose number can easily be varied. The *direct_comm_if* of the CPUs is used to signal interrupts from accelerators or indicate the availability of data from the path dispatcher. The components depicted below the SoC bus are more application specific and are related to a certain number of particular data flow models that are mainly controlled via point to point links between them. The issue of programmability of the different module types via traces will be addressed in the subsequent section.

For a maximum reuse of the basic functionalities and a stepwise specialization concerning their interfaces and the execution of traces, the different resource types are derived in the class hierarchy shown in Fig. 7.

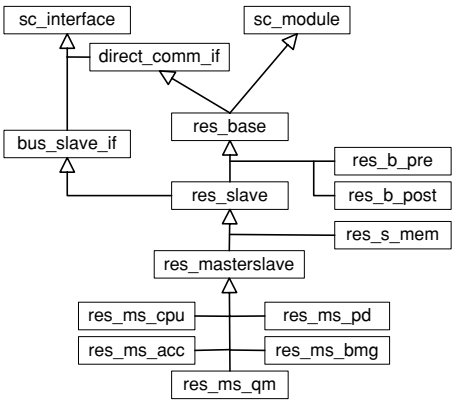

Fig. 7 Class hierarchy of architecture modules

The class *res_base* already contains all necessary functions for reading in the trace specification and for measuring the activity of the module. The derived module type *res_slave* in turn implements additionally the operations of a bus slave and a trace execution process limited to trace primitives relevant to a bus slave. The next extension is then a combined master and slave (*res_masterslave*) that has the full capabilities of trace execution. The CPUs depicted in Fig. 6 actually are of type *res_masterslave*, however, the slave properties are deactivated in the final derivation of the CPU class. In general, the specific properties of the different module types are finally adjusted in the leaves of the class hierarchy. The implementations of the interface functions in the particular resource types allow controlling the interaction between resources and the bus respectively, and triggering the execution of the required traces realized as concurrent processes.

A modular approach is followed for constructing the HW structure of the current simulation model dynamically at elaboration time, taking into account the supported communication infrastructure and the associated SystemC interfaces. The actual settings to be used for the simulation are determined via an XML system configuration file that is read in and interpreted at simulation startup. The file contains sections defining the system architecture and its resources, the measurement and logging data to be output during simulation and the traffic stimuli. This concept allows for an easy adaptation of the model to different architecture

```
<architecture>
      <basic>
         <physical_ports>4</physical_ports>
         <external_link_rate>1000</external_link_rate>
         <clock>200</clock>
      </basic>
      <resources>
         <cpu_modules>3</cpu_modules>
         <acc_modules>1</acc_modules>
         <mem_modules>2</mem_modules>
         ...
      </resources>
      <cpu>
         <clock>500</clock>
      </cpu>
      <bus>
         <type>abstract</type>
         <clock>100</clock>
         <width>64</width>
      </bus>
      ...
</architecture>
<measurement>
      <interval>100000</interval>
      <resource_types><cpu/><acc/><mem/></resource_types>
      ...
</measurement>
<logging>
      <bus>arbitration</bus>
      ...
</logging>
<traffic>
      ...
</traffic>
```

Fig. 8 Sample system configuration file

configurations simply by changing parameters of the configuration file. Figure 8 shows the main parts of a sample configuration file.

The <*architecture*> section specifies the system architecture and all its resources with their parameters, including number and speed of the input ports, base clock frequency and all resource specific parameters. In deviation from the general system clock defined in the <*basic*> section, special clock frequencies can be adjusted for CPUs, accelerators and the bus. This allows scaling all latencies concerning both data transfer and processing in the system. The size of queues contained in some of the resources is also specified in the corresponding sections of the system configuration file.

The <*measurement*> tag determines which simulation data should be captured in a simulation results file for later evaluation. Different levels of detail can be measured per resource type. This encompasses load values of resources including the bus, averaged over a certain time interval, latencies of packets and processing times consumed in different resources or fill levels of queues contained in the system. It is also possible to activate different levels of verbosity messages for observing the processing within the system, e.g. for debugging purposes. The section delimited by the <*traffic*> tag determines the traffic stimuli for the simulation. It may either contain the description of artificial traffic patterns to be generated during simulation or the designation of preprocessed traffic files.

For each type of resource the required number of instances is generated and connected with each other or with the bus. A specific numbering scheme is used to unambiguously identify the different modules of the system architecture, what is necessary for both the establishment of the internal connections and the specification of the target modules as part of the trace definitions. According to the module numbers the bus model directs the transfer request to the correct *sc_port*<> instance in the vector of bus ports. For modeling the interplay of the modules in the ingress and egress data path of the architecture, a data flow model is used that is partially incorporated in the method definitions of the *direct_comm_if* interfaces and thus provides less flexibility concerning trace execution.

For bus communication, a concept has been chosen that allows the usage of bus models on different levels of abstraction. In addition to an abstract, timed model, a cycle-accurate model has been developed that can be used for a more detailed investigation of bottlenecks in the system. For this purpose specific wrappers are available that adapt the high level models of bus masters and slaves to the detailed bus. Both bus models capture the behavior of a CoreConnect PLB bus including a priority-based arbitration scheme, pipelined address phases, split transactions and concurrent transfers on the parallel read and write busses. Bus locks are not supported in both models, while timeouts are covered only in the cycle accurate version. Common transaction level models are located on a higher level of abstraction where these effects are frequently disregarded, leading to lower precision of simulation results.

4.2. System functionality

The functional flexibility is given by the trace-based approach as described in the previous section. Each individual resource instance used in the system architecture requires a list of traces specifying its functionality. On the instantiation of each architecture module a file containing the traces is read and verified, that are later on executed during simulation. Thus, the system functionality can be modified just by changing the trace files, i.e. without the need to make any modification of the SystemC source code of the simulation model.

A module trace is composed of an arbitrary long sequence of trace primitives that is delimited by the End-of-Trace primitive. Table 1 contains an overview of all trace primitives that are currently used for specifying the functionalities of the different resource types. In

Table 1 List of trace primitives

Code	Trace primitive	Type	Parameters
BRS	Bus Read specific size	C	Data amount, target, trace
BRV	Bus Read variable size	M	Reference to packet info
DRS	Dcomm Read specific size	C	Data amount, trace
DRV	Dcomm Read variable size	M	Reference to packet info
BWS	Bus Write specific size	C	Data amount, target, trace, set_semaphore
BWV	Bus Write variable size	M	Reference to packet info
DWS	Dcomm Write specific size	C	Data amount, trace
DWV	Dcomm Write variable size	M	Reference to packet info
DEL	Processing	P	Latency
INT	Issue Interrupt	C	CPU number, interrupt service routine
SEM	Semaphore	P	Slave number
EOT	End of trace	–	–

the third column the type of the primitive is indicated, C for communication primitive, P for processing and M for Macro. Not all primitives are allowed in each resource type. This depends for example on the attachment to the communication architecture or the role as bus master or slave.

The latency value of the processing primitive represents the duration of a certain processing subtask, specified as the number of instructions. The actual time that the resource is using in a SystemC *wait()* statement is then dynamically calculated taking into account the current clock frequency and the number of clock periods that are needed per instruction. Currently, processing latencies are fixed values contained in the trace definition, i.e. all packets being processed experience the same delay. Variable processing latencies needed for capturing data dependencies could easily be implemented by an additional primitive that takes the instruction count from to the annotated information accompanying each packet.

The communication primitives specify both the target module and the trace to be executed there. They also designate the amount of data that would be transferred in the real system, a figure that is used in the communication infrastructure to calculate transfer latencies. The token identifying the packet currently being processed is part of the trace primitive as well. Read and write operations may be denoted as blocking or non-blocking. Communication primitives are translated at simulation run-time into a call of an interface function. In case of a bus-based transaction the bus model with its protocols is activated. If arbitration was successful the trace is triggered in the slave module after the bus transfer time. Otherwise the initiating master will wait for a certain back-off time and then retry the transaction. In case of a transaction over a *direct_comm_if* the method in the target module is directly activated.

The variable size primitives have been defined with respect to our main application area packet processing, where variable sized packets have to be transferred to and from the memory. They are actually macros that retrieve the data amount via an indirect reference to the annotated information of a packet. Furthermore, they take into account the memory architecture and the data structures that are used for storing packets. Packets are assumed to be saved as a variable number of fixed-size data segments that are linked together in a linked list. Data segments and pointers may be mapped onto different memories, e.g. in external SDRAM and in on-chip SRAM respectively. Therefore, at simulation run-time translation functions map the macros to specific sequences of communication primitives that correspond to the memory accesses needed for storing or retrieving the current packet. Different packet storage strategies can thus be investigated by exchanging the translation functions used in

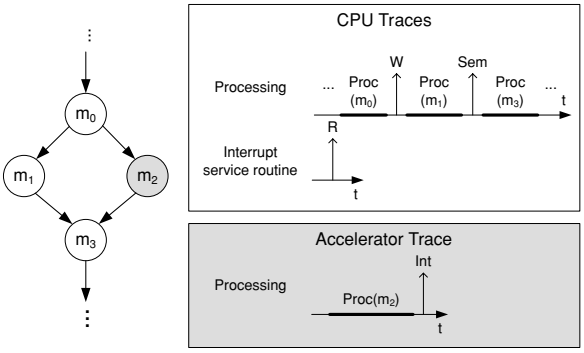

Fig. 9 Usage of semaphore trace primitive and data dependencies

the simulator. Moreover, it is easy to replace these translation functions for application scenarios with specific memory management strategies. Variable sized data transfers without packet segmentation and linkage can be supplemented in the same way as data dependent processing.

Interrupts can be used in accelerators and on some of the modules in the ingress data path to cause CPUs to execute a specific trace as interrupt service routine, e.g. for reading back data or fetching packets for processing. The CPU number as parameter of this primitive determines on which *d_comm* port the interrupt will be issued; a further parameter designates the interrupt service routine to be executed. If an interrupt is called during normal execution of a trace, the trace execution is suspended and resumed after finishing the interrupt trace.

The semaphore trace primitive is of special use to model data dependencies when performing parallel tasks. This is shown in the task graph fragment shown in Fig. 9. Task m_2 is assumed to be offloaded from the CPU and executed in an acceleration module. Further, task m_3 depends on the results of m_2, i.e. it cannot be processed in the CPU unless the results from the accelerator are available.

Thus, tasks m_0, m_1 and m_3 are part of the CPU's processing trace, whereas m_2 is contained in an accelerator trace that ends with an interrupt. In order to enforce the data dependency during simulation, a semaphore trace primitive is inserted in the CPU processing trace before the primitive corresponding to task m_3. When the CPU calls the accelerator, the semaphore is marked active. Conversely, it is deactivated after processing the CPU's interrupt service routine trace, which is triggered by the last element of the accelerator trace. If trace execution in the CPU reaches the semaphore still being in active status, trace processing will be suspended until the semaphore is deactivated. However, if the interrupt has been processed before trace execution in the CPU reaches the semaphore no suspension will occur. This mechanism supports one outstanding call per accelerator and per CPU. Figure 10 shows on the left hand side the contents of the trace files for both the CPU and the accelerator corresponding to the example of Fig. 9. The "Rd trace" of the accelerator is executed before the actual bus transfer and can be used for modeling internal read latencies.

The value of 32 contained in the primitives of the CPU traces identifies the accelerator according to the applied numbering scheme. The third integers in the "W" and "ISR" lines designate the trace to be executed in the accelerator, the final '1' in the bus write command states that the semaphore should be set for the accelerator.

If, as an alternative architecture, the accelerator should be avoided and task m2 be mapped to the CPU, the only changes to the system setup would be the modification of the CPU trace file as shown on the right hand side of Fig. 10 and the modification of the number of accelerators in the system configuration file.

Example with Accelerator

CPU Trace File

```
#number of traces
2
# start of traces
...
DEL 200        # m0
BWS 20 32 0 1 # W
DEL 300        # m1
SEM 32
DEL 150        # m3
...
EOT
BRS 4 32 1     # ISR
EOT
```

Accelerator Trace File

```
#number of traces
2
# start of traces
DEL 40 # m2
EOT
DEL 1  # Rd trace
EOT
```

Example without Accelerator

CPU Trace File

```
#number of traces
1
# start of traces
...
DEL 200      # m0
DEL 500      # m1
DEL 300      # m2
DEL 150      # m3
...
EOT
```

Fig. 10 Trace file examples

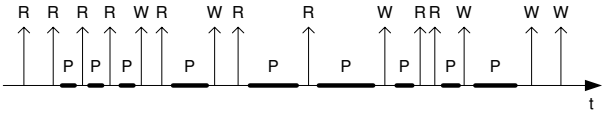

Fig. 11 CPU trace with without cache

The influence of processor caches in respect to memory accesses is currently captured manually during trace definition. Considering the locality of the memory accesses is feasible to a reasonable extent for our specific network application scenarios. A trace for an embedded processor without cache could look like the trace shown in Fig. 11.

Interaction between modules via *direct_comm_if* is less general than in the case of the bus-based transactions. Examples contained in Fig. 6 are the data transfer from the MAC to both pre-processor and buffer manager that may start only if both blocks are ready, or the backpressure mechanism to the queue manager in the egress data path. Consequently, the implementation of the interface functions in each of these more specialized modules has to be tuned to the respective model of data flow that they belong to.

Modeling of the system workload is very application specific. In our case the stimuli are made up of a sequence of arriving packets for each input port. The characteristics are mainly the interarrival times of the packets, the information what trace has initially to be triggered and some other data that has to be annotated like its size or some other packet specific information that is accessible in the simulator via the packet token. Packet stimuli are read by the source module from traffic files, individually for each port, and forwarded to the MAC according to the specified timing. Such a traffic file can be considered as a particular type of trace that constitutes the external triggers of the required functionality. Traffic files may be generated artificially with specific characteristics like packet size or packet rate, in order to investigate the system's sensitivity to these properties. However, the system can be simulated under real world workload as well. For this purpose, traffic that has been recorded in real networks and is available in *pcap* format can be preprocessed and stored in appropriate traffic files.

4.3. Simulation environment

For a systematic investigation of the system performance and its dependency on certain parameters, additional helper tools have been developed. Figure 12 shows an overview of

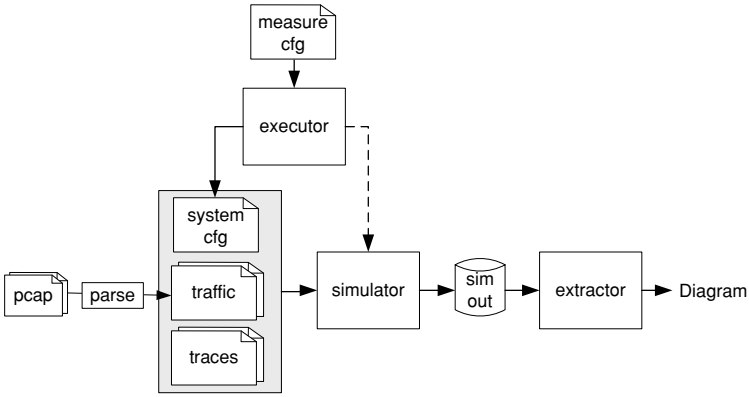

Fig. 12 Simulation environment with helper tools

the current simulation environment that is part of the performance evaluation block depicted in Fig. 1. The main components of the environment are the simulator that implements the TAPES approach described hitherto, the executor that is used for simulation automation and the extractor program for preparing the results.

On simulation startup the simulator reads in the input files marked grey in Fig. 12: The system configuration file containing the complete specification of the system to be simulated and of all outputs that have to be generated; the trace files with the functional description of the model, one for each of the architecture resources; the traffic stimuli for each input port if preprocessed *pcap* traffic files should be used. Then the architecture simulation is carried out and the measured data are written in a simulation output file. Such a simulation generates results for one particular configuration of the system.

In order to investigate the influence of specific architecture or workload parameters on the system performance, the executor is used to step through the desired range of parameter values and to iteratively carry out the corresponding simulations. The executor is controlled by a measurement configuration file, which is essentially an extended version of the system configuration file. It determines the complete configuration of a basic system architecture and those parameters that have to be varied in a series of simulations. Based on this specification the executor repeatedly generates valid system configurations for the corresponding simulation points and starts the simulator. Moreover, it allows the execution of batch jobs each consisting of such a sequence of simulations, enabling extensive investigations without the need for user intervention.

Finally, we use an extractor program that helps in the analysis of the simulation results. It is aligned with the simulation capabilities of the executor and can evaluate individual or series of simulations. According to the user's input requirements, it parses the simulation output, extracts the selected measurement results and prepares them for presentation.

5. Experiments

The following experiments are taken from the network processor domain and encompass the exploration of a network processor SoC architecture using a common SoC bus and standard components as shown in Fig. 2. The path dispatcher, pre-processor and post-processor being more specialized modules are disregarded in this study. Starting from a pure

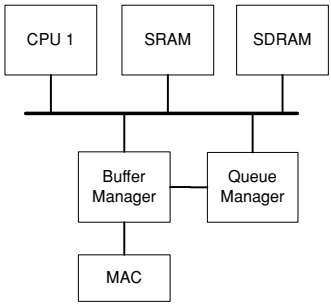

Fig. 13 Basic NP architecture

SW-based solution we explore different measures for performance improvement like adding an additional CPU, increasing clock frequencies, adding a specific DMA engine or modifying the memory architecture. This is accompanied by a bottleneck analysis of the architecture resources. All simulations carried out in the sequel are using the abstract PLB bus model. Figure 13 shows the corresponding architecture that is stepwise modified in the following.

It is assumed that the architecture has to be tailored to an IPv4 forwarding application with the following data flow in the system: Packets are first stored in the memory architecture in segmented way. Depending on the length of the packet and the size of the segments a corresponding number of segments have to be stored in memory and linked together using linked lists. In the course of the exploration we will study different alternatives for the storage of packet segments and linked lists, either in off-chip SDRAM or in on-chip SRAM or in a combination of both. The actual packet processing is carried out in the CPU: First, the packet descriptor is fetched from the buffer manager and then the packet header is read from memory. After processing, modified data is written back and packet descriptors are sent to the queue manager, which determines when packets have to be retrieved from memory and sent out to the target network interface.

The stimulations are done with uniform traffic consisting of equally sized packets with equidistant interarrival times which are determined by the offered input data rate for each port. We assume identical input packet streams on 4 parallel Gigabit Ethernet interfaces with an equal distribution of the destination port of the packets. The packet length is one of the parameters whose influence on the architecture will be studied during the exploration. Unless otherwise specified, CPUs work with a frequency of 500 MHz and need 1.4 clock cycles per instruction. The processing scenario for IP forwarding [15] is modeled with 400 CPU instructions per packet, leading to a consumption of 560 clock cycles.

The memory architectures used in the first exploration steps consist of an external SDRAM memory for the storage of packet data and an on-chip SRAM block for the linked lists. We start with a pure SW solution where processing and packet reception/transmission including the associated memory management are completely done by the CPU without DMA support. I.e. in this configuration we neither have a buffer manager nor a queue manager and the MACs are directly attached to the bus. The CPU fetches only packets from a MAC if the previous packet is completely processed; the CPU is not interrupted by new packets that have been stored in the MAC receive buffer.

Figure 14 shows the output data rate (Mbps) and the packet throughput (Mpps) as a function of the input data rate and the offered packet load with the packet size as parameter.

The results show that the throughput of this solution gets into saturation between 200 Mbps (64 bytes) and 400 Mbps (1500 bytes). While large packets achieve an absolute higher data rate, it can be seen that the throughput in terms of processed packets per second grows with

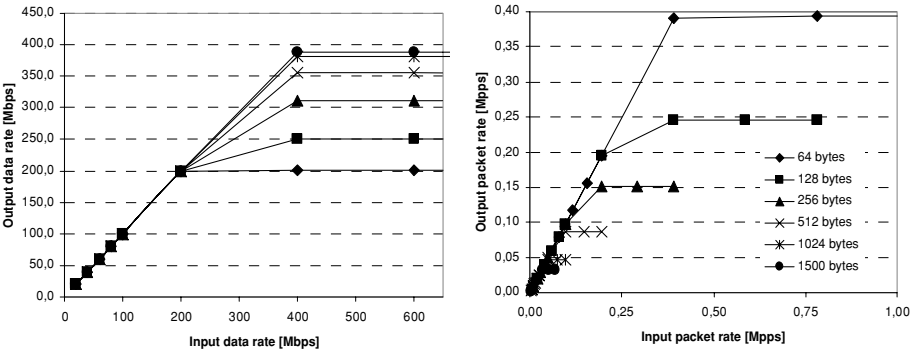

Fig. 14 Performance of pure SW-based solution (1 CPU, 500 MHz)

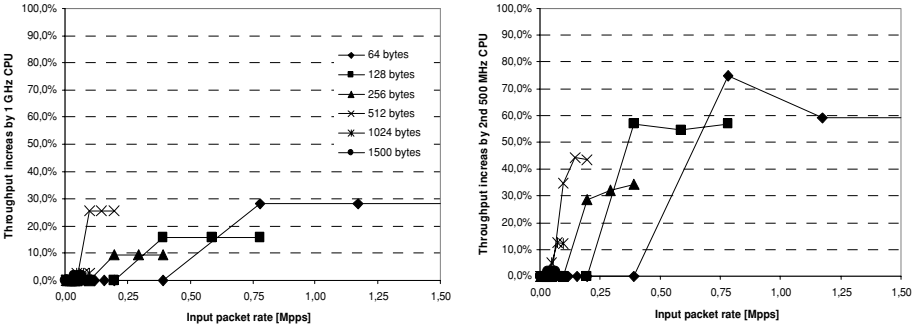

Fig. 15 Performance increase by 1 GHz CPU (left) and a second 500 MHz CPU (right)

decreasing packet size. This is caused by the effort for packet reception and transmission that is proportional to the packet size, thus limiting the budget remaining for the actual packet processing.

Now we try to increase performance by two alternative measures, either doubling the CPU clock frequency to 1 GHz or providing a second 500 MHz CPU. Figure 15 shows the results of these modifications relative to the base architecture.

The benefit from the 1 GHz CPU is very small because only the very limited share of CPU time spent for actual processing can profit from the increased clock speed. The lion's share of instructions is required for data transfers with the MAC and the memory that are not accelerated from this measure. This applies especially for long packets. The solution with two CPUs allows independent processing and data transfers on the PLB bus with its two separate read and write buses, enabling higher performance. However, with increasing packet sizes the probability of contention on the shared resources for the transfers is rising and thus reducing the performance gain from the additional processor.

Now we turn back to the single 500 MHz CPU, however, offload it from the packet reception and transmission tasks including the segment management by introducing buffer and queue managers in the system architecture. The buffer manager autonomously receives and transmits packets without CPU involvement so that the CPU is only responsible for the pure packet processing task. Furthermore, this module contains a queue for packets that have been stored in memory and are ready for processing.

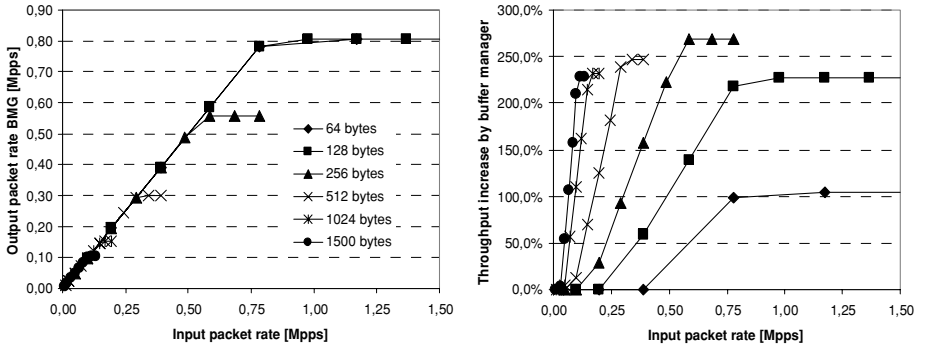

Fig. 16 Performance of architecture with buffer and queue managers (left); relation of throughput of buffer manager enhanced architecture to pure SW solution (right)

The left diagram of Fig. 16 shows that throughput of this configuration is by a factor of two to three higher compared to the reference architecture. The maximum throughput is the same for 64 and 128 byte packets. For longer packets, however, the corresponding figure is significantly lower. The diagram on the right hand side shows the resulting performance increase as a function of the offered load. An accelerating effect is noticeable as soon as the pure SW solution saturates, reaching its peak when the enhanced architecture runs into overload as well. The maximum acceleration is much higher for long packets compared to 64 byte packets because the associated memory management effort is contributing significantly to the processor load in the pure SW solution. We can clearly state that intelligent DMA (buffer and queue managers) helps substantially to improve packet processing performance without loosing flexibility.

As the CPU processing effort is independent of the packet length, similar maximum throughput values for all packet lengths would be expected. However, the results for packet sizes of 256 and more bytes are significantly lower than for 64 and 128 bytes. This observation gives evidence that the CPU is not the bottleneck in these cases. In order to find the reason for this behavior further data are extracted from the simulation log files that can help explain the effect and give clues for further improvements of the architecture. In Fig. 17 the load values from CPU and SDRAM memory block are shown as a function of the sum input data rate of all 4 ports.

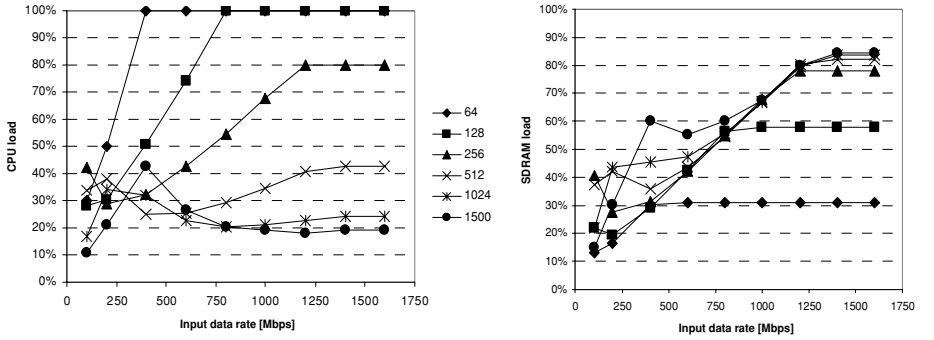

Fig. 17 Load values of CPU (left) and SDRAM (right)

Table 2 Simulation run-times

	Simulation run-times in sec (45,000 packets per entry)					
Bytes	pure SW	pure SW 2 CPUs	pure SW 1 GHz	Buffer mgr.	Buffer mgr. SRAM	Buffer mgr. 2 CPUs
64	184	218	196	119	83	122
128	310	369	322	173	147	168
256	574	682	593	260	252	273
512	1,098	1,308	1,115	454	418	448
1,024	2,140	2,597	2,169	815	752	836
1,500	3,152	3,878	3,160	1,196	1,116	1,185
Σ	7,458	9,052	7,555	3,017	2,768	3,032

The left diagram confirms that the CPU is not the bottleneck for the traffic with packet sizes of 256 bytes or more, in fact it gets significantly idle with the big packets. The reason for this effect can be found in the SDRAM load, shown on the right hand side, that runs into saturation as a consequence of the higher memory bandwidth requirements associated with long packets.

We now want to challenge these bottlenecks by simulating two alternative architectures: In the first experiment the available memory bandwidth is raised by using a pure SRAM-based memory architecture, i.e. administrative as well as packet data is stored in SRAM. The second alternative is increasing processing capacity by introducing a second CPU. In the former case we would expect that the throughput for longer packets saturates at a higher value, in the latter experiment the maximum value for short packets should improve.

The two diagrams shown in Fig. 18 confirm these assumptions: In the SRAM-based architecture all measurements lie on the same curve that ends in the saturation of the CPU. The memory bottleneck is not noticeable any more. In the dual CPU architecture the maximum performance for 64 bytes is doubled, the throughput for 128 byte packets is now limited by the memory bandwidth. Note that the saturation values for the bigger packets are identical to the single CPU architecture, as was expected.

For each simulation point shown in the diagrams above, 5,000 packets have been simulated, making up 45,000 packets for each packet size curve and a total of 270,000 packets per diagram. Table 2 shows the simulation run-times for the different packet sizes carried out for each experiment. The simulations have been performed on an IBM ×Series 346 Server with

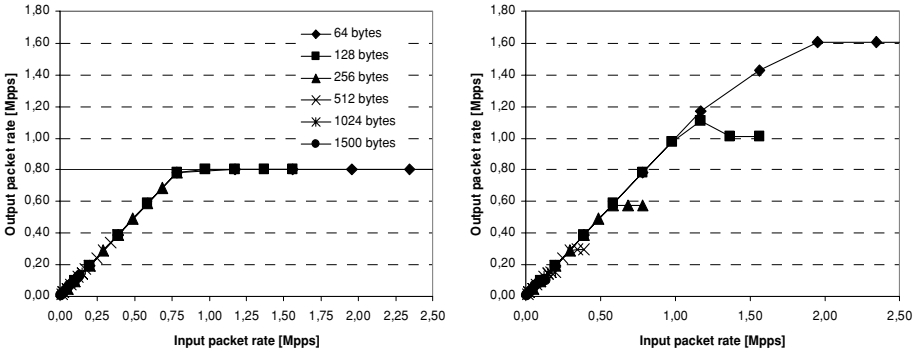

Fig. 18 Performance of a pure SRAM-based architecture (left) and a dual CPU solution (right)

a Xeon 3.2 GHz Processor under Linux Kernel 2.6.13. Note that the simulation software including the SystemC library (Version 2.0.1) has not been compiled for this processor architecture neither was it optimized.

It has to be pointed out that the modifications of the system architecture carried out during the above exploration implied mainly the change of configuration parameters of our simulation model and a new simulation run. The introduction of the buffer and queue managers into the originally pure SW-based architecture required also the removal of four trace elements in the CPU trace specification and the modification of two trace elements in that of the buffer manger. Besides these minor changes that are possible with very low effort no further modifications were necessary in the simulation model.

6. Conclusions and outlook

In this paper we have presented TAPES (Trace-based Architecture Performance Evaluation with SystemC) as an efficient approach for performance evaluation as part of the architecture exploration process. The method uses a trace driven simulation technique and is based on SystemC transaction level modeling. Describing the system functionality on an abstract level and capturing the interaction of the system resources enables a flexible and nevertheless precise investigation of SoC architectures. A special feature of the concept is the low effort to modify the underlying model by changing system configuration and trace files, which describe the hardware architecture and the functionality respectively. As there is no need to modify the associated SystemC code and due to the functional abstraction short turnaround times can be achieved. We have demonstrated the efficient use of the TAPES concept with the exploration of a network processor architecture. Nevertheless, the approach is generic in a way that it can also be applied in different application areas other than packet processing.

Future work will be the support of more heterogeneous communication architectures with several buses or NoCs and of SoC architectures that are dynamically adaptable to failure and load conditions at run-time. A major extension will also be the realization of a GUI that further eases configuration and measurement evaluation.

Acknowledgments We would like to thank Daniel Llorente from TUM for his contributions concerning modeling memory subsystems in TAPES and Christian Sauer from Infineon Technologies for the close cooperation in the SmartFlow project.

This work has partially been funded by the Bavarian Ministry of Economic Affairs, Infrastructure, Transport and Technology under grant IuK178/001 (SmartFlow).

References

1. Balarin, F. et al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.
2. Balarin, F. et al. Metropolis, An Integrated Electronic System Design Environment. *IEEE Computer*, April 2003.
3. Buck, J. et al. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *Int'l J. Computer Simulation*, Apr. 1994.
4. Cai, L. and D. Gajski. Transaction Level Modeling: An Overview, CODES+ISSS, 2003.
5. Donlin, A. Transaction Level Modeling: Flows and Use Models, CODES+ISSS, 2004.
6. Grötker, T., S. Liao, G. Martin, and S. Swan. *System Design with SystemC*, Kluwer Academic Publishers, Boston, May 2002.
7. Henia, R., A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis—the SymTA/S Approach. *IEE Proc.-Comput. Tech*., March 2005.

8. Keutzer, K., et al. System Level Design: Orthogonalization of Concerns and Platform-based Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19, 2000.
9. Kohler, E. et al. The Click Modular Router. *ACM Trans. on Computer Systems*, 18(3), 2000.
10. Lahiri, A., A. Raghunathan, and S. Dey. Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures, ICCAD, 1999.
11. Lieverse, P., P. van der Wolf, and E. Deprettere. A Trace Transformation Technique for Communication Refinement, *CODES*, 2001.
12. Ohlendorf, R., A. Herkersdorf, and T. Wild. FlexPath NP—A Network Processor Concept with Application-Driven Flexible Processing Paths, CODES+ISSS, 2005.
13. Paulin, P., C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design&Test of Computers*, Nov.-Dec. 2002.
14. Prete, A., G. Prina, and L. Ricciardi. A Trace-Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor Systems. *IEEE Trans. Parallel and Distributed Systems*, 6(9), 1995.
15. Ramaswamy, R. and T. Wolf. PacketBench: A Tool for Workload Characterization of Network Processing. *IEEE Workshop on Workload Characterization*, Oct. 2003.
16. Sherman, S.W. and J.C. Browne. Trace Driven Modeling: Review and Overview. In *Proceeding Symp. On Simulation of Computer Systems*, 1973.
17. Sonntag, S., M. Gries, and C. Sauer. SystemQ: A Queuing-Based Approach to Architecture Performance Evaluation with SystemC. In *Proceeding Emb. Comp. Systems: Architectures, Modeling, and Simulation* (SAMOS), 2005.
18. SystemC homepage, http://www.systemc.org.
19. Thiele, L. and E. Wandeler. Performance Analysis of Distributed Embedded Systems, in R. Zurawski (ed.), *Embedded Systems Handbook*, CRC Press, 2005.