A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms

Torsten Kempf, Malte Doerper,
R. Leupers, G. Ascheid, H. Meyr
Institute for Integrated Signal Processing Systems
Aachen University of Technology, Germany
Torsten.Kempf@iss.rwth-aachen.de

Tim Kogel, Bart Vanthournout
CoWare, Inc.
Leuven, Belgium
http://www.CoWare.com
Tim.Kogel@CoWare.com

Abstract

Heterogeneous Multi-Processor SoC platforms bear the potential to optimize conflicting performance, flexibility and energy efficiency constraints as imposed by demanding signal processing and networking applications. However, in order to take advantage of the available processing and communication resources, an optimal mapping of the application tasks onto the platform resources is of crucial importance.

In this paper, we propose a SystemC-based simulation framework, which enables the quantitative evaluation of application-to-platform mappings by means of an executable performance model. Key element of our approach is a configurable event-driven Virtual Processing Unit to capture the timing behavior of multi-processor/multi-threaded MP-SoC platforms. The framework features an XML-based declarative construction mechanism of the performance model to significantly accelerate the navigation in large design spaces.

The capabilities of the proposed framework in terms of design space exploration is presented by a case study of a commercially available MP-SoC platform for networking applications. Focussing on the application to architecture mapping, our introduced framework highlights the potential for optimization of an efficient design space exploration environment.

1. Introduction

One of the most challenging tasks in modern System-on-Chip design projects is to map a complex application onto a heterogeneous architecture in adherence to the specified performance and cost requirements. The effective performance of processing elements is often confined by the communication architecture, since memory access latency does not keep pace with the increasing computational power. General purpose processors resolve the memory access bottleneck by using sophisticated cache and memory hierarchies. Unfortunately this approach is often not applicable for embedded applications due to the poor memory locality of stream driven and packet based data processing.

Instead, embedded processor architectures are increasingly equipped with Hardware Multi-Threading (HW-MT) [1] to perform task switches with virtually no performance overhead. By that, the application inherent Task Level Parallelism (TLP) is exploited with the purpose of hiding the memory access latency. This effectively leads to a significant increase in processor utilization. The HW-MT technique is already widely employed in the network processor domain [2] but recently finds its way into advanced multimedia [3] and wireless signal processing platforms [4].

Beside the immediate benefit of increased utilization, HW-MT can be considered as a lean operating system implemented in hardware to efficiently share the processing resources among multiple concurrent tasks. In analogy with today's software operating systems (SW-OS), the HW-MT concept bears the potential to bring a disciplined management of processing resources to the data processing domain. From the perspective of the functional tasks, this 'processing management' introduces a virtualization of the computational resources. This virtualization of the architectural elements represents an efficient concept to cope with the complexity of MP-SoC platforms: now the system architect can *allocate* processing and communication resources in a deterministic way [5].

However, the spatial and temporal mapping of the functional tasks to processing elements as well as the mapping of the inter-task data exchange to a communication architecture while meeting performance and cost requirements is an unresolved challenge. In this paper we propose a SystemC based simulation framework, which enables the system architect to evaluate arbitrary task mappings by creating an abstract and yet sufficiently accurate *performance model* of the considered application together with the anticipated architecture.

The application is first represented as a set of untimed reactive SystemC tasks communicating through an unified Transaction Level Modeling (TLM) [6] interface. Next the processing requirements of each individual task are characterized by annotating the delay budgets to the communication events. Finally we introduce the concept of a Virtual Processing Unit (VPU) to capture the impact of shared processing elements to the SoCs performance. As an intermediate layer between the timed task network and the underlying event driven simulation kernel, the major purpose of the VPU is to compute timing of the multi-threaded task execution under the consideration of task swapping and preemption.

The major benefit of the outlined approach is the *declarative construction* of the performance model: the individual timing annotations as well as the mapping of the tasks onto the respective processing elements is specified by means of an eXtended Markup Language (XML) description. This generic mapping mechanism significantly shortens the iterative exploration cycle as well as improves model reuseability.

After the subsequent discussion of related work, we present the envisioned MP-SoC design flow and give an intuitive introduction of the task model. The following section 4 introduces the VPU concept and defines in detail the operational semantic. The value of our approach is highlighted by a large scale case-study in the context of the networking application domain.

2. Related Work

System Level Design is considered the appropriate way to deal with the ever increasing complexity and heterogene-

1530-1591/05 $20.00 © 2005 IEEE

ity of SoC architectures [7]. Various actor-oriented frameworks are proposed to capture arbitrary Models of Computation (MoC) for the purpose of system level modeling and tool supported paths to exploration, implementation and/or verification [6, 8, 9]. The modeling strategy presented in this paper can be implemented on top of any of these MoC generic frameworks. We selected SystemC mainly because of the broad user acceptance and commercial tool support.

Complementary to our top-down refinement flow, the Component Based Design paradigm [10] advocates the bottom-up platform composition from a parameterizable IP library, containing off-the-shelf processing elements, communication fabrics and hardware dependent Software layers. This approach is clearly advantageous for the rapid exploration and implementation of the general purpose portion of the application [11], whereas our approach is focused on application specific architectures executing the data-processing part.

The highest possible abstraction level for design space exploration and application mapping is static performance analysis [12, 13]. Other approaches are closer related to simulation frameworks for top-down exploration and refinement like ARTEMIS [14], MESH [15], STepNP [16] and work on abstract RTOS modeling [17, 18].

The ARTEMIS project [14] is focused on an automated refinement of coarse-grain Kahn Process Network algorithm models to fine-grain architecture models for the purpose of design space exploration and synthesis.

Similar to our approach, the Modeling Environment for Software and Hardware (MESH) [15] project is concerned with modeling of heterogeneous MP-SoC platforms above the cycle-level accuracy. Here, schedulers are considered as the central modeling element to capture the dynamic and data-dependent nature of MP-SoC platform mapping.

The SystemC based STepNP [16] simulation framework also advocates the joint consideration of communication architecture and application specific processing elements. STepNP is focused on the early integration of Instruction-Set Simulators (ISS) into cycle-level TLM platform models [19], whereas in our approach processing elements and interconnect are situated on a higher packet-level TLM abstraction layer.

Madsen et al. propose the combined modeling of NoC and RTOS scheduling at a yet higher abstraction level, where the application tasks are only represented as set of timing budgets for processing and communication without any functional information [17].

Work on RTOS modeling [18] and generation [20] employs similar modeling techniques as our approach, but particularly addresses the selection and configuration of the Software RTOS for general purpose processing.

Our modeling framework addresses quantitative performance analysis during the early conceptualization of highly complex Network-on-Chip (NoC) enabled MP-SoC platforms. The unique transparent mapping mechanism presented in this paper combines the flexibility of non-functional performance analysis with an accuracy within few percent of fully cycle accurate TLM simulation [21].

3. System Level Design Flow

We first give an overview of the complete flow, before we introduce the design space exploration framework. Then we motivate the concept of task modeling, where we introduce the Virtual Architecture Mapping methodology to capture the effect of task execution on dedicated hardware as well as on single-threaded processor cores.

3.1. Design Flow Overview

The overall flow follows the multi-level SoC design strategy proposed by Magarshack and Paulin [22], which separates MP-SoC development into four distinctive phases.

- The *functional phase* deals with development of Hardware independent Software and application specific algorithms. We assume, that this phase also comprises the partitioning of the application into a set of loosely coupled functional blocks and the extraction of Task Level Parallelism (TLP). For the performance relevant data processing portion of typical signal processing and networking applications, this architecture independent partitioning is mostly straight forward and can be immediately derived from the algorithmic block diagram.
- The *MP-SoC platform phase* covers the system-architecture specification by integration of high level IP blocks, along with the spatial and temporal mapping of the application to the MP-SoC platform. We particularly address this phase, which is concerned with the full functional and architectural complexity of MP-SoC platforms.

Additionally, *high-level IP creation* and *basic IP creation* are concerned with the development of the individual SoC components like e.g. embedded processors or interconnect technologies. These two phases are not in the scope of this paper.

3.2. MP-SoC Mapping Phase

As depicted in figure 1, our MP-SoC framework follows the well-known y-chart principle [23], where a set of functional application models is merged with a set of architecture models in a dedicated mapping step. As an extension of the general y-chart paradigm, in our case the timing is separately specified during the mapping phase to keep it independent from both the functional model as well as the architecture model. This mechanism is the key to enhance the reusability of architecture and application models as well as the flexibility during design space exploration. In reference to the flexible and highly abstracted mapping mechanism, the developed embodiment of the y-chart principle is called *Virtual Architecture Mapping (VAM)*.

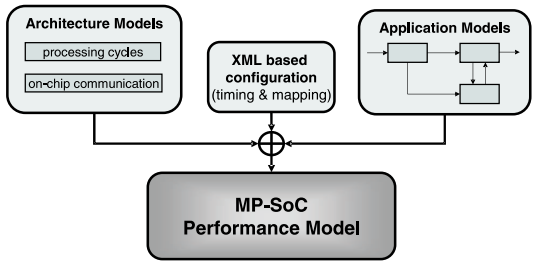


Figure 1. Virtual Architecture Mapping

In our previous work we have conceived a well defined packet-level TLM paradigm [24] for efficient modeling of embedded applications and the anticipated communication architecture. The available set of generic, parameterizable on-chip communication models cover shared buses as well as full scale on-chip networks [25]. The simulation environment also contains a comprehensive set of analysis tools for functional and performance validation.

One key aspect for efficient design space exploration is a declarative specification mechanism, i.e. the following aspects of the MP-SoC architecture are defined by an XML based configuration file, which contains:

- the configuration of the timing model (basically the number of required cycles per task execution),
- the number of available processors and number of supported concurrent threads per processor,
- the mapping of the application tasks to processors and threads,
- the instantiation, parameterization and interconnection of the communication nodes,
- the instantiation and address mapping of the memory architecture.

During the elaboration phase of the simulation, the architecture specification is extracted from the configuration files and bound to the application model by means of Virtual Architecture Mapping. During the simulation run, evaluation modules connected to the architecture models collect and aggregate statistical information like resource utilization, latency, and throughput. On completion of the simulation, this statistical information is visualized by means of histogram and communication graph views. Based on the collected data, the system architect may modify the MP-SoC architecture and/or the application mapping until the requirements are met.

3.3. Task Modeling

This paragraph introduces the task modeling. Tasks are represented by their pure functionality and an individual timing model. The functionality is pure C/C++ code, whereas the timing model influences only the notification time of externally visible events. These annotated times characterize the architecture implementation of the task and are derived from the required processing time of the anticipated processing element to execute the task's functionality.

Following to the notation scheme of the Tagged Signal Model (TSM) [26] we will now introduce a formal representation of the timing annotation. After some fundamental definitions, the modeling of a functional process is derived.

Elementary Definitions: An **event e** consists of a time tag $t \in \mathcal{T}$ and a value $v \in \mathcal{V}_{ADT}$. In our packet-level TLM paradigm, a value is represented as an Abstract Data Type (ADT), which is basically a C++ class object. Predefined members of this ADT are a priority, a delay and a state. The tag and the value fields of the event are accessible by the point operator, i.e. $e_i.value.priority$ denotes the $priority$ field of event e_i. A **signal s** is a set of events, which can be viewed as a subset of $\mathcal{T} \times \mathcal{V}$.

In the following considerations functional SystemC processes are represented as **timed Communicating Extended Finite State Machines (tCEFSM)** to reason about the timing annotation and mapping mechanism. A tCEFSM is a 7-tuple $(\mathcal{I}, \mathcal{O}, \mathcal{Z}, f, \mathcal{U}, \mathcal{D}_{busy}, \mathcal{D}_{delay})$ with

- a set of input events $\mathcal{I} \subseteq \mathcal{T} \times \mathcal{V}_{ADT}$ and output events $\mathcal{O} \subseteq \mathcal{T} \times \mathcal{V}_{ADT}$
- a finite, non-empty set of explicit states $\mathcal{Z}$.
- a set of variables $\mathcal{U} = (u_1, \ldots)$, which represent the implicit state.
- a state transition function $f : \mathcal{Z}^* \times \mathcal{I} \mapsto \mathcal{Z}^* \times \mathcal{O}$, where $\mathcal{Z}^*$ denotes the set of all implicit and explicit states
- a set of busy periods $\mathcal{D}_{busy} = \{\Delta t_{busy,i}\}$
- a set of processing delays $\mathcal{D}_{delay} = \{\Delta t_{i,d}\}$

The tCEFSM is activated on the arrival of a new input event and instantaneously responds with a state transition and possibly the generation of one or more output events. To model the busy time of the task during the processing of the state transition, the next activation is not possible before the time Δt_{busy} has elapsed as illustrated in the upper part of figure 2. Additionally the generated events are projected into the future to account for the period $\Delta t_{i,d}$ between task activation and event generation. Note that this scheme is sufficiently expressive to capture the notion of pipelined processing elements with $\Delta t_{busy} < \Delta t_{i,d}$, i.e. the processing of a new input event starts before the generation of the result.

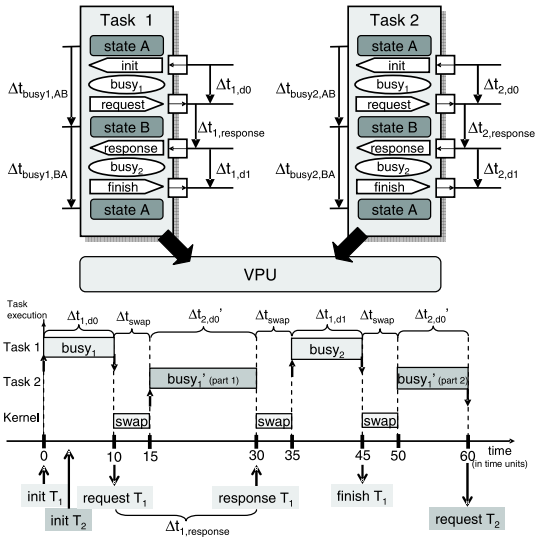


Figure 2. VPU Performance Model

By guarding the activation of the tCEFSM during the busy periods and delaying the generated events, the externally visible behavior of the tCEFSM corresponds to the task execution on the anticipated processing element. Compared to non-functional task representation by means of processing budgets, our coarse-grain annotation of the functional SystemC processes yields a very accurate model of the timing characteristic. Additionally the simulation speed is significantly faster than on a target instruction set simulator.

This modeling methodology enables mapping of functional tasks to single-threaded processors as well as to dedicated hardware blocks. In the following section we introduce the concept of a Virtual Processing Unit (VPU) to enhance the introduced methodology to model shared processing resources.

4. VPU

We will now present the Virtual Processing Unit concept. First we introduce an intuitive introduction of the functionality and in the following paragraph the operational semantic of this generic SystemC model is derived.

4.1. VPU Introduction

The VPU enables the system architect to investigate the mapping of the application tasks with respect to space and time. Spatial mapping denotes the assignment of a task to one of the physical processing elements in the MP-SoC platform. Temporal mapping refers to the allocation of a time budget (derived from the number of processing cycles) on this particular processing element. Note that the task itself remains untouched and no recompilation is required to explore the design space, as the total simulation environment is configured by an XML description during the simulations initialization phase.

An example depicted in figure 2 illustrates the timing annotation and VPU mapping mechanism. The upper part of figure 2 shows two tasks and their individual timing characteristic, which are spatially mapped to a single VPU instance. The bottom of figure 2 shows the resulting timing in response to an assumed scenario. In the following we discuss the event sequence to illustrate the impact of the VPU:

First task 1 is activated by the external $init\ T_1$ event, executes the first portion of its functionality, and generates the external $request\ T_1$ event after 10 time units . In the meantime, the activation event $init\ T_2$ has already occurred, but task 2 cannot start execution before the first task is finished

and swapped out. The VPU takes this additional delay into account and notifies the external event to activate the execution of task 2 at the correct time. In the given scenario, the communication request from task 1 returns before task 2 has finished the first portion of its functionality. Since task 1 is configured to have a higher priority, task 2 is preempted and not resumed before task 1 has completed its functionality. The request generated by the functionality of task 2 is delayed by the additional preemption time, thus the externally visible event of this request is notified at the corresponding time of concurrent task execution.

Of course, the new task mapping capabilities are compliant with our existing communication models. As outlined in the previous chapter, the complete MP-SoC platform phase is now supported by a versatile exploration framework.

4.2. Operational Semantic

According to the Tagged Signal Model (TSM) we introduce a formal representation of the timing annotation and the VPU mapping mechanism.

The concept of a VPU generalizes the modeling of a SW-OS as well as HW-MT processing elements. Following the Virtual Architecture Mapping (VAM) mechanism, the VPU mapping is achieved by manipulation of the externally visible event tags. Before the incorporation of any specific Real Time Operating System (RTOS), our transparent mapping scheme and parameterizable VPU enables the efficient design space exploration without modifying of the functional tasks.

By mapping different functional tasks represented as timed CEFSMs to one VPU, the VPU guards the activation of all tasks against incoming events as illustrated in figure 3. In order to calculate the tags of all incoming and generated events, the VPU maintains two data structures:

The **Priority Queue** $\mathcal{U}_{PQ}$ is a list of events $e_i \in \mathcal{T} \times \mathcal{V}$, which stores incoming events. By that the individual busy times $\Delta t_{busy,i}$ of the associated tasks are taken into account. The associated method $schedule\_processes(\mathcal{U}_{PQ})$ schedules the next active task from the list of pending events in the Priority Queue $\mathcal{U}_{PQ}$ without removing the event from the queue. An additional method $remove\_finished\_processes(\mathcal{U}_{PQ})$ removes the activating event of a finished task from the Priority Queue $\mathcal{U}_{PQ}$.

A **Delay Queue** $\mathcal{U}_{DQ}$ is a list of events $e_i \in \mathcal{T} \times \mathcal{V}$, which projects generated events into the future according to their tags. The event tags are calculated with respect to the individual delay annotations $\Delta t_{delay,i}$.

Additionally the VPU calculates the delay penalty of task swapping and task preemption. Along these lines we define a VPU to be a 7-tuple

$$\mathcal{P}_{VPU} = (\mathcal{S}_I, \mathcal{S}_O, \mathcal{E}_{Internal}, \mathcal{U}_{PQ}, \mathcal{U}_{DQ}, \mathcal{U}_{VPU}, \overline{f}_{VPU})$$

- a set of input signals $\mathcal{S}_I = \mathcal{S}_{VPU,CI} \cup \mathcal{S}_{VPU,FI}$, where $\mathcal{S}_{VPU,CI}$ denotes a set of input signals connected to the on-chip communication network and $\mathcal{S}_{VPU,FI}$ denotes a set of input signals connected to the associated functional tasks.
- a set of output signals $\mathcal{S}_O = \mathcal{S}_{VPU,CO} \cup \mathcal{S}_{VPU,FO}$, where $\mathcal{S}_{VPU,CO}$ denotes a set of output signals connected to the on-chip communication network and $\mathcal{S}_{VPU,FO}$ denotes a set of output signals connected to the associated functional tasks.
- a set of internal events $\mathcal{E}_{Internal} = \{ e_{PQ,update}, e_{DQ,update} \}$
- a Priority Queue $\mathcal{U}_{PQ} \subseteq \mathcal{T} \in \mathcal{V}$ and a Delay Queue $\mathcal{U}_{DQ} \subseteq \mathcal{T} \in \mathcal{V}$
- a set of internal variables $\mathcal{U}_{VPU} = \{ u_{busy}, u_{priority}, \Delta t_{swap} \}$, where
  - a state variable u_{busy}, which is initialized as false
  - a variable $u_{priority}$ retains the priority of the active process
  - a swapping time Δt_{swap} depending on the VPU's status
- a set of functions $\overline{f}_{VPU}$:

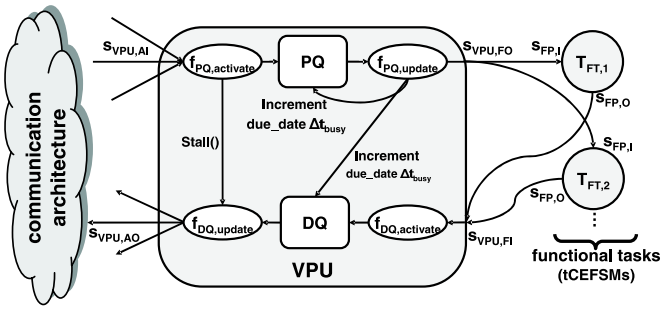


Figure 3. Virtual Processing Unit (VPU)

- $f_{PQ\_activate}$ being sensitive to external events on the signals $\mathcal{S}_{VPU,CI}$ connected to the communication network
- f_{PQ_update} being sensitive to the internal event $e_{PQ,update}$
- $f_{DQ\_activate}$ being sensitive to events on the signals $\mathcal{S}_{VPU,FI}$ connected to the associated functional tasks
- f_{DQ_update} being sensitive to the internal event $e_{DQ,update}$

In the following we elaborate the operational semantics of the VPU functions and illustrate the work-flow of the VPU model by means of the example depicted in figure 3.

Events arriving on the incoming signals $\mathcal{S}_{VPU,CO}$ activate the function $f_{PQ\_activate}$. This function inserts the arriving event into the Priority Queue $\mathcal{U}_{PQ}$ and in case of preemption or idle state the VPU directly activates the further processing of the arrived event.

```
function f_PQ_activate{
  U_PQ.insert(e_VPU,AI);
  if((u_priority < e_VPU,AI.value.priority) || (!u_busy)))
    e_PQ,update.notify();
}
```

The function f_{VPU,PQ_update} handles the tag manipulation of both pending events for activation of the mapped tasks as well as outgoing events generated by the tasks.

```
0  function f_PQ_update{
1    remove_finished_processes(U_PQ);
2    if(U_PQ.notEmpty()) {
3      if(e_tmp = schedule_process(U_PQ)) {
4        if(e_tmp.value.state == init) {
5          e_VPU,FO = e_tmp;
6          nbr_cycles = e_VPU,FO.notify(Δt_swap);
7          Δt_busy = nbr_cycles * clock_period
8          e_tmp.value.state = busy;
9          e_tmp.value.due_date = now + Δt_busy + Δt_swap;
10         u_init = true;
11       }
12       if(!u_busy) {
13         u_busy = true; u_priority = e_tmp.value.priority;
14       } else {
15         if((u_priority < e_tmp.value.priority) || (u_init)) {
16           // preemption
17           for((all events in PQ)&&(state == busy))
18             event.value.due_date += Δt_busy + Δt_swap;
19           for(all events in DQ)
20             event.value.due_date += Δt_busy + Δt_swap;
21           u_priority = e_tmp.value.priority;
22           u_init = false;
23         } else { // resume task
24           for((all events in PQ)&& (state == busy))
25             event.value.due_date += Δt_swap;
26           for(all events in DQ)
27             event.value.due_date += Δt_swap;
28           Δt_busy = e_tmp.value.due_date - now;
29           u_priority = e_tmp.value.priority;
30         }
31       }
32       e_PQ,update.notify(Δt_busy + Δt_swap);
33     }
```

```
34      } else {
35          u_busy = false;  u_priority = −1;
36      }
37  }
```

On every execution of f_{VPU,PQ_update} with a non-empty priority queue, the VPU checks whether a new task needs to be scheduled from the set of pending events in the Priority Queue $\mathcal{U}_{PQ}$ (line 3). In case an event has been scheduled for the first time (line 4), the delayed activation of the functional task takes the penalty for task swapping into account (line 6 and 7).

Recall that the activation of a functional task always returns immediately and the VPU performs the required timing manipulation of the events. The product of the return value nbr_cycles and the VPUs $clock\_period$ denotes the individual timing annotation of the activated task (line 7). At the same time the new events generated during the task activation are inserted into the Delay Queue $\mathcal{U}_{DQ}$. These events remain inside this queue until their tag is due. As depicted in figure 3, sending of the projected events is handled by the functions $f_{DQ,update}$ and $f_{DQ,activate}$.

Task preemption occurs when the current task has a lower priority than the selected task. In this case all generated events in the Delay Queue $\mathcal{U}_{DQ}$ and the tags of already activated processes in the Priority Queue $\mathcal{U}_{PQ}$ have to be delayed (lines 17 – 20). The preemption time is calculated from the busy time Δt_{busy} of the displacing processes and the required swapping time Δt_{swap}.

Figure 2 illustrates a typical execution sequence in case of preemption. At time 30 the response of task 1 displaces the execution of task 2. Originally event $request\ T_2$ would occur at time $15 + \Delta t_{2,d0} = 15 + 25 = 40$. Instead, the event is delayed by $\Delta t_{1,d1} + \Delta t_{swap} = 15$ time units. Resuming task 2 at time 45 causes another Δt_{swap} tag increment of event $request\ T_2$, which finally occurs at time 60.

Coming back to the discussion of f_{VPU,PQ_update}, the tag incrementation of already generated events in case of task resuming is performed in lines 24 – 27. Finally the next activation of f_{VPU,PQ_update} occurs after the current task is finished, i.e. after the busy time of the active process Δt_{busy} plus the swapping penalty Δt_{swap} (line 32). If no events are pending in the priority queue, the VPU switches to idle state and waits for the arrival of new events (line 35).

As already mentioned the functions $f_{DQ,update}$ and $f_{DQ,activate}$ maintain the Delay Queue $\mathcal{U}_{DQ}$, where:

- The function $f_{DQ,activate}$ inserts the incoming events at the functional signal into the Delay Queue $\mathcal{U}_{DQ}$.
- The function $f_{DQ,update}$ manages the sending of due events to the architectural signals of the VPU.

In summary, the VPU enables the transparent mapping of multiple functional tasks onto a shared processing resource. The configurable Δt_{swap} parameter models of the impact on performance during task swapping of a slow SW-OS compared to hardware supported HW-MT. The modular implementation of the framework allows the customizing the task scheduling algorithm by overloading the function $schedule\_process(\mathcal{U}_{PQ})$. This enables the incorporation and early investigation of the RTOS scheduling policy.

5. NPU Case Study

To demonstrate the fidelity of the outlined approach, this section presents the results of a design project accomplished with the introduced design methodology. We have selected an IPv4 forwarding application with Quality of Service (QoS) functionality and demonstrate the framework's exploration capabilities during a case study of a state-of-the-art Network Processing Unit (NPU) platform.

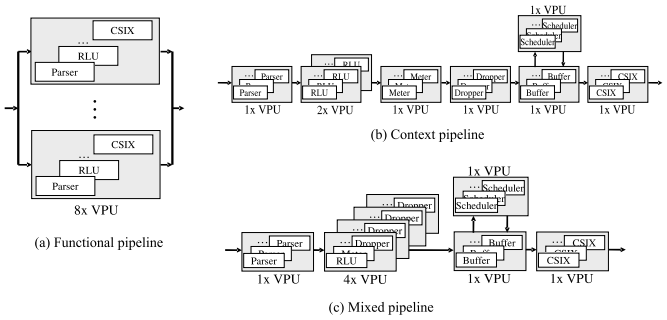


Figure 4. Software Pipelines

5.1. Reference Architecture

The capability of the introduced methodology has been evaluated in the context of the Intel IXP2400 NPU [27]. The task characterizations in terms of memory accesses as well as computation cycles are taken from the original Intel documentation [2] and are illustrated in table 1.

The reference architecture of Intel consists of 8 RISC-like processing elements each implementing 8 concurrent hardware threads. These processors have been modeled as VPUs, where each VPU instantiates 8 functional tasks to model the 8 threads of each processor. Since HW-MT processors swap tasks within a single clock cycle the VPU swap time Δt_{swap} is one cycle. The communication network of the IXP2400 architecture comprises 7 exclusive next-neighbor connections and 3 global shared buses. To achieve the required OC-48 performance (corresponding to 2.5 Gbit/s) the timing budget to process a minimum size 48 Byte packet is only 147ns.

In this section, we illustrate the exploration capabilities of our framework, focussing on the task mapping of the IPv4 application to the Intel architecture.

5.2. Design Space Exploration

The mapping of the application tasks onto various VPU configurations reveals the impact of processing element features like execution speed and HW-MT support. To achieve a fair comparison between the considered task mappings, the processing and communication requirements of all functional tasks are tied to the values in table 1. All VPUs are modeled with a frequency of 600 MHz.

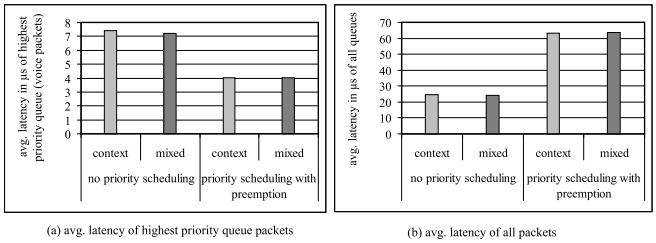
Three alternative spatial task mappings depicted in figure 4 are evaluated: according to the functional software pipeline (figure 4a) each VPU concurrently executes the whole application. Second, figure 4b shows a context software pipeline, where each VPU concurrently executes 8 instances of the same task. The third option depicted in figure 4c represents a mixed functional/context pipeline.

line	func. task	busy cycles (Δt_{busy})	SRAM accesses	DRAM accesses	no. alloc. tasks
0	Parser	70	3	2	8
1	RLU	160	10	1	24
2	Meter	80	2	0	4
3	Dropper	80	2	0	4
4	Buffer	≈0	4	0	8
5	Scheduler	100	0	0	4
6	CSIX	80	3	1	8

Table 1. Reference Mapping

Our traffic scenario is a typical IP traffic profile with an effective data rate of 2.06 Gbit/s. Investigation of the three introduced software pipelines shows, that the functional pipeline falls short to achieve the throughput requirements, as a throughput of appr. 1.89 Gbit/s is reached. Therefore in the following discussion the functional pipeline will be omitted, whereas the context and mixed pipeline will be further investigated.

The impact of temporal task mapping is examined by means of the context- and the mixed-pipeline systems and two different VPU configurations. First a non priority based simulation and second a priority based simulation is evaluated. Depending on the QoS class of each received IPv4 packet the priority is determined, e.g. video and voice packets are of high priority, whereas flooding packets are of low priority. This scheduling mechanism influences the temporal execution of the different tasks mapped to each VPU. Preferring higher priority packets, task preemption as well as task stalling occurs to speed up the processing of high priority packets.



(a) avg. latency of highest priority queue packets

(b) avg. latency of all packets

Figure 5. Latency Measurements

The simulation results depicted in figure 5 show, that priority based task scheduling in the VPUs achieves a significant reduction of approximately 40% in the latency of high priority voice packets. As a tradeoff the average latency of all processed packets rises by a factor of 2.6. These tradeoff illustrates the crucial impact of this kind of system architecture decisions on the system performace.

In summary the results of the case-study demonstrate the capabilities of the simulation framework in terms of flexibility for design space exploration as well as high simulation speed. As soon as the SystemC model of the application is available a single engineer can carry out the task mapping experiments within a few hours or days, as only configuration files have to be modified. The simulation speed of this highly complex MP-SoC architecture model for IP forwarding is in the range of 100,000 cycles per second on a 2.0 GHz Linux host, which is roughly 2 orders of magnitude faster than the ISS based simulator in the Intel Software Development Kit [27].

6. Conclusion

We propose a system level simulation framework for early investigation of MP-SoC platform architectures in the context of the application. The major contribution of this paper is a highly flexible timing annotation and task mapping mechanism to capture the performance impact of single- and multithreaded processing elements. Here the concept of a Virtual Processing Unit enables the rapid exploration of spatial and temporal application mappings to arbitrarily complex multiprocessor platforms with no design overhead.

The major advantage of our modeling approach is the combination of high simulation speed, modeling efficiency and accuracy to rapidly evaluate architectural alternatives. Together with the Network-on-Chip simulation environment [25] the resulting MP-SoC framework covers the exploration of the complete design space spread by multiple heterogeneous processing elements and complex communication architectures.

Our exploration framework has been successfully applied to the customization of an MP-SoC platform, which performs IP forwarding with Quality of Service support. This complex case study illustrates the potential of the exploration capabilities of the developed framework.

Our future work will focus on the incorporation of RTOS specific services and the additional deployment of analytical analysis- and optimization-techniques.

References

[1] T. Ungerer, B. Robic, J. Silc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.

[2] S. Lakshmanamurthy, K.-Y. Liu, Y. Pun, L. Huston, U. Naik. Network Processor Performance Analysis Methodology. *Intel Technology Journal*, 6(3), August 2002.

[3] M.J. Rutten, J.T.J. van Eijndhoven, E.G.T. Jaspers, P. van der Wolf, O.P. Gangwal, A. Timmer, E.-J.D. Pol. A Heterogeneous Multiprocessor Architecture for Flexible Media Processing. *IEEE Design & Test of Computers*, 19(5):39–50, July-August 2002.

[4] C. J. Glossner, T. Raja, E. Hokenek, M. Moudgill. A Multithreaded Processor Architecture for SDR. *Proc. of the Korean Institute of Communication Sciences*, 19(11):70–85, November 2002.

[5] T. Agerwala. Systems Trends and their Impact on Future Microprocessor Design. Keynote of 35th Annual International Symposium on Microarchitecture, November 2002.

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

[7] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[8] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[9] D. Gajski, J. Zhu, R. Dömer et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[10] M.-A. Dziri, W. Cesrio, F.R. Wagner, A.A. Jerraya. Unified Component Integration Flow for Multi-Processor SoC Design and Validation. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2004.

[11] W. Cesario, A. Baghdadi, L. Gauthier et al. Component-Based Design Approach for Multicore SoCs. In *Proc. of the Design Automation Conference (DAC)*, 2002.

[12] M. Jersak, R. Henia, R. Ernst. Context-Aware Performance Analysis for Efficient Embedded System Design. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2004.

[13] T. Pop, P. Eles, Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *"Proc. Int. Symp. on Hardware/Software Codesign (CODES)"*, 2002.

[14] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P. van der Wolf, E.F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[15] J.M. Paul, A. Bobrek, J.E. Nelson, J.J. Pieper, D.E. Thomas. Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[16] E. Bensoudane P.G. Paulin, C. Pilkington. Stepnp: A system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6):17–26, Nov-Dec 2002.

[17] Jan Madsen, Shankar Mahadevan, Kashif Virk, and Mercury Gonzalez. Network-on-chip modeling for system-level multiprocessor simulation. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium RTSS03*, pages 82–92, December 2003.

[18] A. Gerstlauer, H. Yu, D.D. Gajski. RTOS Modeling for System Level Design. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2003.

[19] D. Quinn, B. Lavigueur1,G. Bois, M. Aboulhamid. A System Level Exploration Platform and Methodology for Network Applications Based on Configurable Processors. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2004.

[20] V. J. Mooney, D. M. Blough. A Hardware-Software Real-Time Operating System Framework for SoC's. *IEEE Design & Test of Computers*, 19(6):44–51, Nov/Dec 2002.

[21] M, Ariyamparambath, D. Bussaglia, B. Reinkemeier, T. Kogel, T. Kempf. A Highly Efficient Modeling Style for Heterogeneous Bus Architectures. In *"Proc. IEEE Int. Symp. on System-on-Chip (SoC)"*, November 2003.

[22] P. Magarshack, P. Paulin. System-on-chip Beyond the Nanometer Wall. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[23] P. Lieverse, P van der Wolf, E. Deprettere,K Vissers. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. In *"Proc. IEEE Int. Workshop on SIgnal Processing Systems (SIPS)"*, 1997.

[24] T. Kogel, A. Wieferink, R. Leupers, Gerd Ascheid, H. Meyr, D. Bussaglia, M. Ariyamparambath. Virtual Architecture Mapping: A SystemC based Methodology for Architectural Exploration of System-on-Chip Designs. In *"Proc. Int. Workshop on Systems, Architecturs, Modeling and Simulation(SAMOS)"*, July 2003.

[25] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens. A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks. In *CODES+ISSS*, October 2003.

[26] E.A. Lee, A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, Dec 1998.

[27] Intel Network Processors. *http://developer.intel.com/design/network/products/npfamily/*.