Configurable Multiprocessor Platform with RTOS for Distributed Execution of UML 2.0 Designed Applications

Tero Arpinen, Petri Kukkala, Erno Salminen, Marko Hännikäinen, and Timo D. Hämäläinen
Tampere University of Technology, Institute of Digital and Computer Systems
Korkeakoulunkatu 1, FI-33720 Tampere, Finland

Abstract

This paper presents the design and full prototype implementation of a configurable multiprocessor platform that supports distributed execution of applications described in UML 2.0. The platform is comprised of multiple Altera Nios II softcore processors and custom hardware accelerators connected by the Heterogeneous IP Block Interconnection (HIBI) communication architecture. Each processor has a local copy of eCos real-time operating system for the scheduling of multiple application threads. The mapping of a UML application into the proposed platform is presented by distributing a WLAN medium access control protocol onto multiple CPUs. The experiments performed on FPGA show that our approach raises system design to a new level. To our knowledge, this is the first real implementation combining a high-level design flow with a synthesizable platform.

1. Introduction

Disciplined approaches are required when designing increasingly complex digital systems. Design abstraction using layered system model hides the complexity of underlying layers and makes design reuse more efficient.

Common abstraction layers of an embedded platform are presented in Figure 1. A Real-Time Operating System (RTOS) schedules the execution of application threads, and offers services to access available platform resources. Further, a platform Application Programming Interface (API) is a set of functions accessing hardware resources on processing elements. Consequently, this hides the changes in hardware from an application and RTOS. The communication between processing elements is abstracted by a Network-on-Chip architecture.

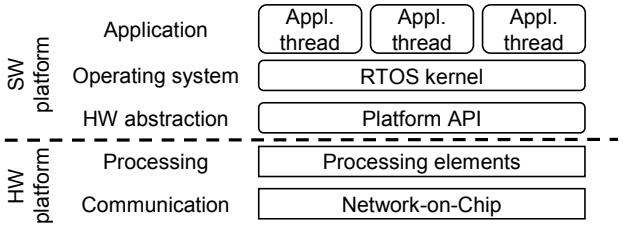

Figure 1. Layered system model.

Unified Modeling Language (UML) is an object-oriented language that is traditionally used to design software systems. The use of UML in the embedded system design requires the use of UML profiles to improve the semantics in this domain. In this work, we exploit TUT-Profile, which is targeted at real-time embedded systems [8]. TUT-Profile supports the design automation from UML to a physical System-on-Chip (SoC) implementation. Applications modelled in UML are platform-independent, which enables the separation of application functionality and hardware. Further, this enables efficient hardware/software co-design and fast prototyping on different hardware platforms.

Figure 2 presents an example of a UML-based SoC design flow with TUT-Profile. Architecture exploration is used for optimizing component allocation, application task mapping and scheduling. The SoC implementation is separated into an application code generation from TUT-Profile *application* and *mapping models*, and platform synthesis based on an *architecture model*.

This paper presents a configurable multiprocessor platform that is designed to support both the execution of UML applications and the automated architecture configuration to govern the platform synthesis from library components. Further, the platform includes a library of software components supporting the distributed execution of UML applications on multiple CPUs. The novel features enable rapid and automated physical implementation from a UML model, which consequently enables efficient verification and evaluation on FPGA.

The components and tools used in this work are listed in Table 1. The hardware platform contains multiple Nios II softcore processors and hardware accelerators. They are

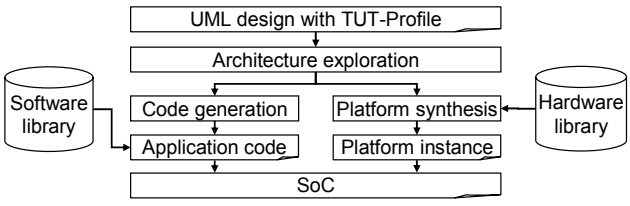

Figure 2. UML-based SoC design flow.

Table 1. Categorization of the components and tools used in the platform development.

Self-made components/tools	Off-the-shelf components/tools
TUT-Profile	Tau G2 UML 2.0 tool
TUTMAC UML model	Quartus II 5.0 suite
UML process distributor tool	Nios II GCC toolset
Architecture configuration tool	eCos RTOS
IPC support functions	State machine scheduler
HIBI API	Nios II softcore CPU
HW accelerator device drivers	Nios development board
HIBI communication arch.	
Nios-to-HIBI DMA	
HW accelerators	

connected by the Heterogeneous IP Block Interconnection (HIBI) [9][11]. The architecture configuration tool automates the platform hardware generation using a library of RTL descriptions of hardware components. The logic synthesis for FPGA is performed using Quartus II tool from Altera.

The platform software includes eCos RTOS [10], which is used for the scheduling of multiple application threads. Inter-Processor Communication (IPC) support functions are required to perform distributed application execution. Device drivers for hardware accelerators are used for accessing common hardware resources. The application software is automatically generated from UML 2.0 models by Telelogic Tau G2 UML 2.0 tool and UML process distributor tool. The latter distributes the execution of an application onto multiple CPUs.

As a case study, a WLAN Medium Access Control (MAC) protocol called TUTMAC [5] is mapped to the multiprocessor platform and prototyped on single FPGA. The experimental results are measured to evaluate the scalability and performance of the platform in practice.

The paper is organized as follows. First, related work is presented in Section 2. Section 3 presents the design of the multiprocessor platform. Mapping a UML application onto the platform is presented in Section 4. Next, Section 5 shows the platform prototyping on FPGA. In Section 6, experimental results are presented according to measurements on FPGA. Finally, Section 7 concludes the paper.

2. Related work

In this paper, we focus on a combination of a loosely-coupled Multiprocessor on a Programmable Chip (MPoPC), RTOS, and distributed UML application mapping procedure. Related work has several variations, of which some of the closest are examined below.

Hung *et al.* present a tightly-coupled MPoPC which uses softcore processors [4]. In the proposal, Symmetric Multiprocessing (SMP) with Nios CPUs is enabled by introducing a custom Cache Coherency Module (CCM). In SMP processing topology, CPUs share the same memory and operating system, and application tasks are assigned between CPUs at run-time. However, this does not suit well with the platform comprised of heterogeneous processing elements.

Wang *et al.* present an application specific MPoPC implementation performing LU-factorization for linear equation matrices [13]. The implementation utilizes six Nios CPUs without an operating system. IPC is implemented via a shared memory.

Takada *et al.* present a custom configurable RTOS implementation for loosely-coupled MPoPC [12]. The RTOS is executed with multiple MicroBlaze CPUs on Virtex-II FPGA. The focus in the multiprocessor RTOS development is in hardware/software co-configuration techniques in order to create scalable RTOS which is suitable for variable multiprocessor configurations.

Automatic code generation from UML model for embedded systems is under active research [2][7]. However, the approaches are targeted for a single processor implementations, and thus, lack the code generation for multiprocessor systems.

Drosos *et al.* present an implementation of an embedded baseband modem terminal for a wireless LAN application on a platform with two ARM CPUs and an FPGA circuit [1]. The system is specified with UML and software parts are created separately for both CPUs with UML automatic code generation tool. Approach assumes fixed platform and mapping. An RTOS is not utilized.

Compared to our approach, the related work lacks the high-level configurability in UML, and the integration and automation of design phases to produce a prototype. Our approach presents a configurable multiprocessor platform, which is seamlessly integrated with the design automation. Consequently, this leads to a seamless design flow from a UML model to a physical implementation.

3. Multiprocessor platform design

The platform is composed of identical Nios II CPU sub-systems comprised of a CPU and peripherals. Each sub-system has its own local instruction and data memory spaces; *there is no shared memory*. IPC and transactions with hardware accelerators are performed using HIBI.

HIBI is a communication architecture developed at the Tampere University of Technology (TUT). IP units connect to HIBI using a wrapper. HIBI allows using several segments and clock domains over bridges. Further, HIBI includes a property to send messages having a higher priority than normal data. In this platform, messages are used to inform the receiver about the size of the transfer prior to the actual transmitted data.

Nios II is a 32-bit RISC softcore processor. There are three core variants differing on the pipelines, caches, and arithmetic logic units. Nios II CPU utilizes Avalon switch fabric to connect with sub-system peripherals. In Avalon, each connected master-slave pair has dedicated wires with each other, leading to a point-to-point connected network.

Each Nios II CPU sub-system includes a Nios II CPU, timer, boot-ROM, dual-port memory, and possibly

instruction and data caches, depending on the core used. In addition, they have a connection to an external instruction/data memory. The boot-ROM includes a unique processor ID to identify the sub-system. A dual-port memory is used to buffer data in HIBI transactions. A custom Nios-to-HIBI (N2H) DMA controller forwards data between HIBI wrapper and dual-port memory. The CPU and N2H are Avalon masters, whereas the timer and memory components are slaves.

Each CPU executes a local copy of eCos RTOS in a local memory and the application threads are mapped to the CPUs at compilation time. HIBI is accessed through the HIBI API defined as a device driver running on a single eCos thread. All other software drivers accessing hardware accelerators operate through the HIBI API.

4. Mapping UML application

We use the TUTMAC protocol as an example application. It is a dynamic reservation Time Division Multiple Access (TDMA) based MAC protocol that supports negotiation of Quality of Service (QoS) parameters for data transfers. Parts of the protocol require real-time guarantees and are computationally intensive, such as TDMA scheduling, Advanced Encryption Standard (AES) function for data encryption, and 32-bit Cyclic Redundancy Check (CRC) for data error detection.

UML statechart diagrams are used for the behavioral modeling of applications. The statecharts implement asynchronously communicating Extended Finite State Machine (EFSM) models [3], and the formalism allows automatic code generation. EFSMs are defined as states and the transitions between them. A state transition is triggered by a received signal or by an expired timer. This is followed by actions, such as variable assignments, operation calls, condition statements, and signal transmissions.

The communication between state machines is carried out with signals. Signals are comprised of a header and payload. The header includes signal ID, sender/receiver addressing and possibly other parameters, such as signal priority. The payload includes signal parameters that can consist of various data types. The signal passing between different state machines is performed via signal queues.

In TUTMAC, the class hierarchy of the application model has been designed using class diagrams. Composite structure diagrams describe the class instances (parts) and connections between their ports. The behavior of the protocol has been described using statechart diagrams combined with the UML textual notation. According to the TUT-profile, the instances of the state machines are called *application processes*. Table 2 presents the key properties of the TUTMAC application model to illustrate the complexity of the protocol.

Although Tau G2 supports automatic code generation from the UML model, it does not support automatic application mapping nor distributed execution of an application on multiple CPUs. These features are covered with our UML process distributor and IPC support functions, respectively. These tools are governed by TUT-Profile. Thus, TUTMAC application, architecture and mapping models are designed according to the profile.

Table 2. The properties of the TUTMAC UML model.

Property	Amount
Class diagrams	18
Composite structure diagrams	5
Statechart diagrams	41
State machines	20
Ports	52
Signal types	40
Signal size (bytes)	0-1550
Generated C code size (lines)	8775

The architecture model is composed by using a library of parameterized models that are particularly defined for different hardware components, such as Nios II and HIBI. An architecture model is presented in Figure 3, in which the platform instance is comprised of four Nios II CPUs and three hardware accelerators, including AES, CRC and radio interface, interconnected by a single HIBI segment.

After the application model and the architecture model have been described, a mapping model is created. The mapping model defines the distribution of application processes between different processing elements. The mapping is performed in two phases. First, the application processes are grouped together to form a set of *process groups*. Second, the formed groups are mapped to certain processing elements. Process groups are used to cluster processes together according to functional relationships with each other, such as local data dependencies and heavy interaction. Also, groups are used to define the threads on RTOS execution. In this, each process group mapped to a Nios II CPU is implemented as a single eCos thread. An example of mapping a set of process groups to the described architecture model is illustrated in Figure 4.

Application process distribution between eCos threads and CPUs can be performed according to different criteria, such as system deadlines, work load division, signal relations between different processes, and the size of processes. An architecture exploration tool, such as

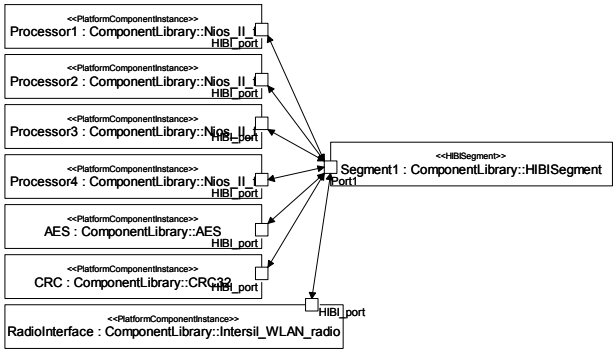

Figure 3. Architecture model of a platform instance.

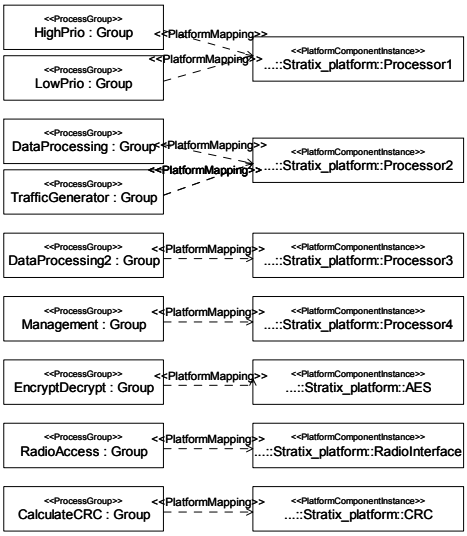

Figure 4. Mapping process groups to processing elements.

Koski [6], can be used to optimize the mapping.

The code generated by Tau G2 supports POSIX, which is exploited to integrate the execution of UML applications with eCos. Based on the created mapping model, UML process distributor tool creates information about which threads are activated on a certain CPU, and attaches this to the generated code. Finally, the codes are compiled to our platform using Nios II GCC toolset.

Currently, all threads are included in the program code of each CPU, but only a part of them are activated on a certain CPU. Memory footprint can be reduced by linking only active processes into executable code of each CPU. The hardware platform is generated by the architecture configuration tool based on the architecture model.

Figure 5 illustrates an example of the application process distribution and signal passing arrangements within a platform instance containing two Nios II CPUs. All processes in the application model are divided into four groups implemented as eCos threads. CPU 1 executes threads 1 and 4 while threads 2 and 3 are inactive. Correspondingly, CPU 2 executes only threads 2 and 3. A state machine scheduler performs synchronization and scheduling of processes within a single eCos thread. Processes located at different eCos threads on the same CPU share the local signal queue.

Signal passing between processes located at different CPUs is carried out as follows. Each CPU has an identical mapping table that indicates activated processes on each CPU. According to this table, IPC functions detect signals that are directed to processes executed on other CPUs. HIBI Tx function is called to send such a signal over HIBI to the correct remote CPU. At the remote CPU, the signal incoming from HIBI is processed and attached to the local signal queue by the HIBI Rx thread. Finally, the signal is forwarded to correct recipient process by the state machine scheduler.

5. Platform implementation on FPGA

The platform prototypes contain 2-6 Nios II CPUs on Stratix II EP2S60 FPGA. The FPGA has 24,176 Adaptive Logic Modules (ALMs) and 2,544,192 bits of on-chip RAM memory. ALMs are the basic building blocks of logic in the Stratix II device family and the ALM count of the FPGA corresponds to 60,440 equivalent 4-input look-up tables. The Nios development board consists of the FPGA and several peripherals, such as an Ethernet controller, RAM and ROM memory devices, UARTs, and prototype expansion headers.

The platform hardware implementation on the development board with 6 Nios II CPUs is depicted in Figure 6. For purposes of our application, Intersil MAC'less Prism HW1151-EVAL WLAN radio is attached to one of the prototype expansion headers. The radio is compatible with the IEEE 802.11b radio standard

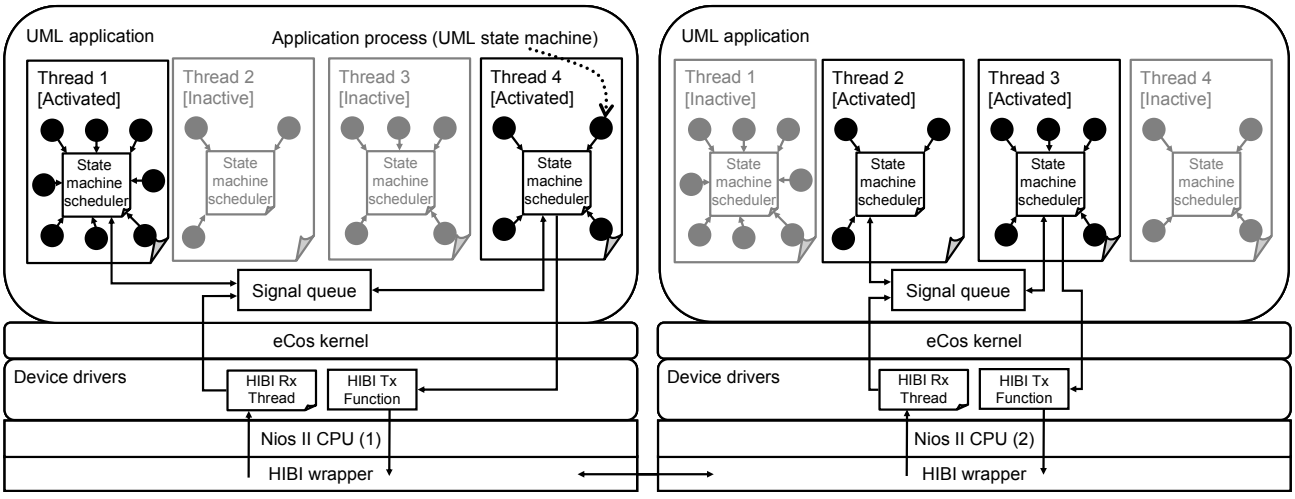

Figure 5. Application process distribution in eCos threads and signal passing arrangements.

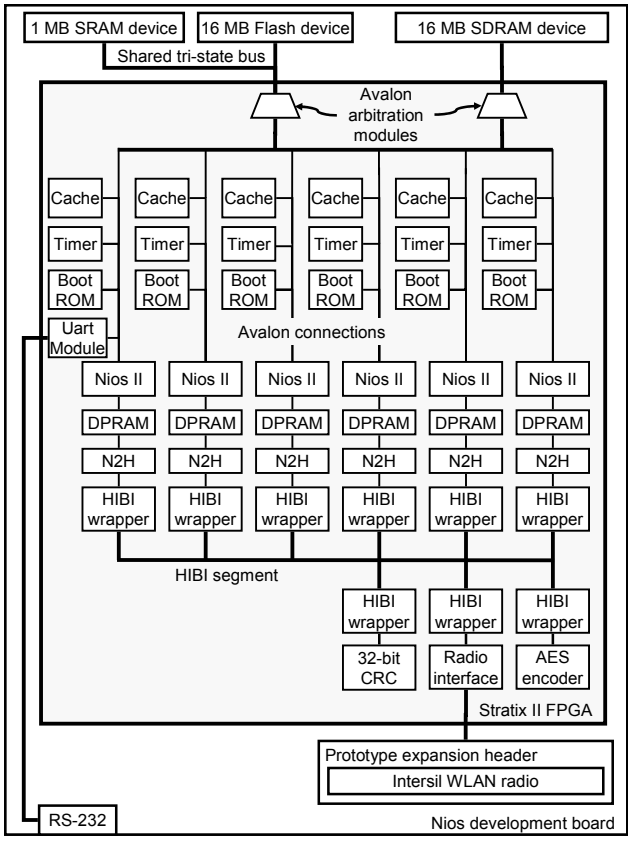

Figure 6. Platform implementation on FPGA.

containing only the physical layer of the standard. The radio is controlled by radio interface logic on FPGA. Hardware accelerated functions include AES encoder and 32-bit CRC calculation unit.

The external memory devices on Nios development board are used due to limited amount of on-chip memory on FPGA. The Nios development board includes 1 MB of SRAM memory and 16 MB of SDRAM memory both acting as Avalon slaves. Their capacities are divided between CPUs for local instruction and data memories. Furthermore, 16 MB of flash memory is used to store the program images for each CPU. The images are copied to run-time memory locations at system reset. It should be noted that each CPU has conceptually a local memory, although, several memories are implemented on the same physical memory device for the prototyping purposes.

Boot-ROM, dual-port, and cache memories are implemented using on-chip memory blocks of the FPGA. Nios II CPUs share the external memory slaves and are thus forced to arbitrate with each other. Avalon and HIBI data buses are implemented as 32-bit wide.

One of the CPUs is assigned as an I/O CPU which gathers debug data from other CPUs via HIBI bus and forwards this data through a serial port to a workstation. This CPU does not participate in the execution of UML application processes. In Figure 6 this CPU is located farthest to the left.

Table 3. Platform resource usage and F_{max}.

Nios II CPUs	Area [ALM]	Area [%]	Memory [bits]	Memory [%]	F_{max} [MHz]
2	11 680	48	372 828	14	78.90
3	14 769	61	501 904	19	68.05
4	17 728	73	630 980	24	59.34
5	20 495	84	760 056	29	53.25
6	22 758	94	889 132	34	48.95

6. Experiments on multiprocessor platform

In the experiments, standard Nios II cores were used with 4 KB of instruction cache. The system resource usage was measured with 2-6 Nios II CPU sub-systems, radio interface, AES, and CRC compiled into the FPGA. Table 3 presents the number of ALMs and on-chip memory bits used as well as the maximum operating frequency (F_{max}) for different platform instances. The critical path is originated from the Avalon arbitrator module of the shared tri-state bus. Moreover, attaching more CPUs to this bus further decreases the F_{max}. 50 MHz system clock frequency was used in FPGA execution.

6.1. Signal passing delays

Delays in signal passing between two application processes were measured in three different scenarios. In the first scenario, the processes resided in the same eCos thread. In the second scenario, the processes resided in different eCos threads on the same CPU and the eCos context switch took place right after the signal output. In the third scenario, the processes resided in different CPUs and signal passing was performed via HIBI.

The measurements were repeated with variable amount of payload lengths attached to the signal. Each signal includes additional header of constant 32 bytes in length. Measurements were performed with two Nios II CPUs both executing UML test application on external SRAM memory device. Figure 7 outlines the measured delays.

The signal passing delay between the processes on different threads was approximately 1.4x compared to intra-thread signal passing. Both were almost constant with variable payload lengths. In contrast, the delay of signal passing between CPUs increased in proportion to the payload size. This is due to data transfers over HIBI. Further, the delay increases more than linearly, since currently, IPC data gets fragmented into small separate transfers. Consequently, this causes frequent interrupt requests issued by the N2H DMA.

6.2. Distributed TUTMAC application statistics

The correct functionality of the distributed TUTMAC application was verified on platform instances with 1-4 CPUs (excluding the I/O CPU). The verification was performed using two development boards as terminal platforms connected together by a radio link, and sending data over the wireless network to both directions. The

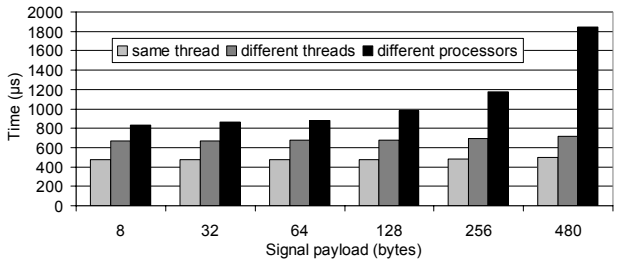

Figure 7. Delays in UML signal passing scenarios.

presented mapping procedure was applied several times for the TUTMAC model, each time manually modifying the process grouping and mapping for each platform instance to optimize the reception delay of the protocol.

The reception delay is one of the key parameters of the protocol. It depicts the time that is consumed by the protocol to process a received packet when data passes through the protocol from radio interface to a user. Based on the reception delay, the theoretical maximum data throughput can be calculated. However, it should be noted that the overall throughput of a wireless network is also dependent on the physical radio link and channel structure of TDMA scheduling.

Figure 8 presents the measured average reception delay and corresponding calculated theoretical maximum throughput as a function of number of CPUs used. The results illustrate that the best result was achieved with two CPUs, leading to 1.44x speed-up over the single-CPU configuration. No further improvement was achieved with three or four CPUs due to increased IPC and small amount of parallel computing in the application. Furthermore, memories of the additional third and fourth CPUs were implemented on the slower SDRAM device, whereas the first two CPUs utilized the faster SRAM device and thus gained higher execution performance.

The required memory footprints of the object codes are listed in Table 4. The presented memory requirement multiplies with the number of CPUs in the platform. eCos takes approximately half of the total allocated memory.

7. Conclusions

This paper presented the design and implementation of a configurable multiprocessor platform, which supports the distributed execution of applications that have been

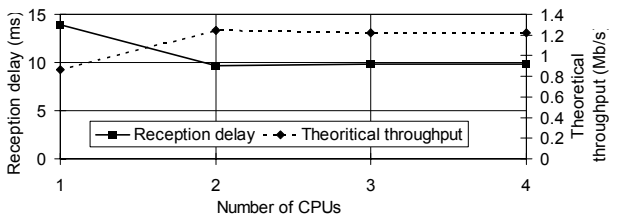

Figure 8. Reception delay and maximum throughput with different number of CPUs.

Table 4. Required memory footprint.

Software part	Code [bytes]	RW-data [bytes]	Total [bytes]
TUTMAC model	21 624	1 900	23 524
eCos	75 920	47 791	123 711
State machine scheduler	29 028	3 049	32 077
HIBI API	7 232	61 824	69 056
Total software	133 804	114 564	248 368

described in UML 2.0. Further, the scalable platform supports the automated platform synthesis, which enables a seamless flow from UML to physical implementation. Experiments with the TUTMAC UML 2.0 design proved that the platform is scalable and the distribution of functionality onto the platform can be managed with the tools and design methodology presented. Future work consists of developing tools enabling dynamic re-mapping of application processes between different threads and CPUs. In addition, shortening IPC delays by improving the functionality of the N2H DMA is under work.

References

[1] C. Drosos *et al*. Hardware-software design and validation framework for wireless LAN modems. In *Proc. Computers and Digital Techniques,* pp. 173-182, Nov. 2004.

[2] M. Edwards and P. Green. *UML for real: UML for Hardware and Software Object Modeling.* Kluwer Academic Publishers, pp. 127-147, May 2003.

[3] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming.* 51(1): 43-75, 2002.

[4] A. Hung, W. Bishop, A. Kennings. Symmetric Multiprocessing on Programmable Chips Made Easy. In *Proc. DATE'05,* pp. 240-245, Mar. 2005.

[5] M. Hännikäinen *et al*. TUTWLAN - QoS Supporting Wireless Network. *Telecommunication Systems - Modelling, Analysis, Design and Management,* 23(3,4):297-333, 2003.

[6] T. Kangas *et al*. UML-Based Multi-Processor SoC Design Framework. Accepted to *Proc. Transactions on Embedded Computing Systems, ACM,* Feb. 2006.

[7] P. Kukkala *et al*. UML 2.0 Implementation of an Embedded WLAN Protocol. In *Proc. PIMRC'04*, pp. 1158-1162, Sept. 2004.

[8] P. Kukkala *et al*. UML 2.0 Profile for Embedded System Design. In *Proc. DATE'05,* pp. 710-715, Mar. 2005.

[9] O. Lehtoranta *et al.* A Parallel MPEG-4 Encoder for FPGA Based Multiprocessor SoC. In *Proc. FPL'05*, pp.1-10, Aug. 2005.

[10] A. J. Massa, *Embedded Software Development with eCos*, Prentice Hall PTR, Nov. 2002.

[11] E. Salminen *et al*. HIBI v.2 Communication Network for System-on-Chip. In *LNCS 3133,* pp. 412-422, Jul. 2004.

[12] H. Takada et al. Hardware/software co-configuration for multiprocessor SoPC (work-in-progress report). In *Proc. WSTFES'03,* pp. 7-8, May 2003.

[13] X. Wang and S.G. Ziavras. Parallel direct solution of linear equations on FPGA-based machines. In *Proc. IPDPS'03,* pp. 22-26, Apr. 2003.