Automated Distribution of UML 2.0 Designed Applications to a Configurable Multiprocessor Platform

Mikko Setälä, Petri Kukkala, Tero Arpinen,
Marko Hännikäinen, and Timo D. Hämäläinen

Tampere University of Technology, Institute of Digital and Computer Systems
P.O. Box 553, FI-33101 Tampere, Finland
mikko.setala@tut.fi

Abstract. This paper presents automated distribution of embedded real-time applications modeled in Unified Modeling Language version 2.0 (UML 2.0). The automated distribution requires methods and tools for design automation, as well as the run-time environment for the distributed execution on the target platform. Executable application code is generated from UML models, and UML with a custom profile is used to abstract hardware architecture and configure application mapping. For experimenting, a full featured WLAN terminal was designed in UML and implemented as a distributed multiprocessor system-on-chip (SoC) on an FPGA prototype platform. Measurements show that a 50-70% reduction in protocol delays is achived with distribution, and delay variations are reduced 45-85%.

1 Introduction

To fulfill the real-time constraints of complex embedded real-time systems, parallelism and heterogeneous multiprocessor architectures are exploited. With complex hardware, distribution of the application functionality onto the different processing elements is a challenging task. To enable the development of hardware independent, reusable software, the hardware implementation should be invisible for the software designer. At the same time, mapping different application tasks to the architecture should be straightforward.

In this paper we present automated distribution of applications described in Unified Modeling Language 2.0 (UML 2.0). The automated distribution consists of methods and tools for design automation, as well as the run-time environment for the distributed execution. The applications are executed on a configurable multiprocessor platform with a Real-Time Operating System (RTOS). The run-time environment enables the communication of processes of the same application executed on different CPUs, while remaining invisible for the application level, as shown in Fig. 1.

Traditionally UML has been used in designing large software systems, but recently it has been emerging also in embedded system design. The publication of the UML 2.0 standard [1] brought several important extensions to the language. Consequently, modern UML modeling tools automatically generate executable code from UML models, making it possible to use a single language for modeling, verification and even implementation of applications. In our approach, UML is also used to create an abstract

S. Vassiliadis et al. (Eds.): SAMOS 2006, LNCS 4017, pp. 27–38, 2006.
© Springer-Verlag Berlin Heidelberg 2006

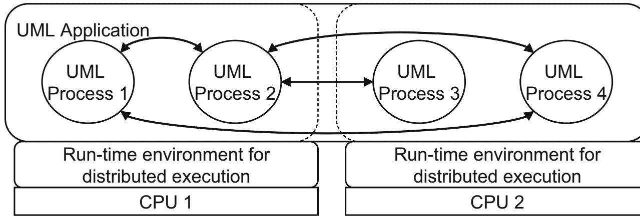

Fig. 1. Communicating processes of a UML application executed on different CPUs

model of hardware architecture and configure application mapping. The implementation of a full featured Medium Access Control (MAC) protocol for Wireless Local Area Networks (WLAN) is presented as a case study to evaluate the feasibility and performance of the approach.

The paper is organized as follows. Chapter 2 presents the related work. The automated implementation flow is presented in Chapter 3. Chapter 4 presents the WLAN protocol case study and the related UML models. The run-time environment is presented in Chapter 5. In Chapter 6 the performance of the implementation is studied. Chapter 7 concludes the work.

2 Related Research

Studies in microprocessor design have shown that a multiprocessor architecture consisting of several simple CPUs can outperform a single CPU using the same area [2], if the application has a large degree of parallelism. Kaiserswerth has analyzed parallelism in communication protocols [3], stating that they are suitable for distributed execution, since they can be parallelized efficiently and also allow for pipelined execution.

A common approach in designing distributed systems is the utilization of *middleware*, such as the Common Object Request Broker Architecture (CORBA) [4], to abstract the underlying hardware implementation from the application level. However, the general middleware implementations are too complex for embedded systems. Thus, several middleware approaches have been developed especially for real-time embedded systems [5] [6] [7].

UML has potential to be used as a design environment for distributed embedded systems, due to its powerful and extensible notations. However, the absence of a standard UML profile for modeling hardware and mapping has slowed down the development of supporting methods and tools. The UML Platform profile proposed in [8] as well as the Embedded UML profile proposed in [9] support the modeling of hardware resources and services, as well as application mapping. Object Management Group (OMG) has specified a UML profile for CORBA, which allows the presentation of CORBA semantics in UML [10].

Born *et al.* have presented a method for the design and development of distributed applications using UML [11]. It uses automatic code generation to create code skeletons for component implementations on a middleware platform. However, direct executable code generation from UML models or modeling of hardware in UML are not utilized.

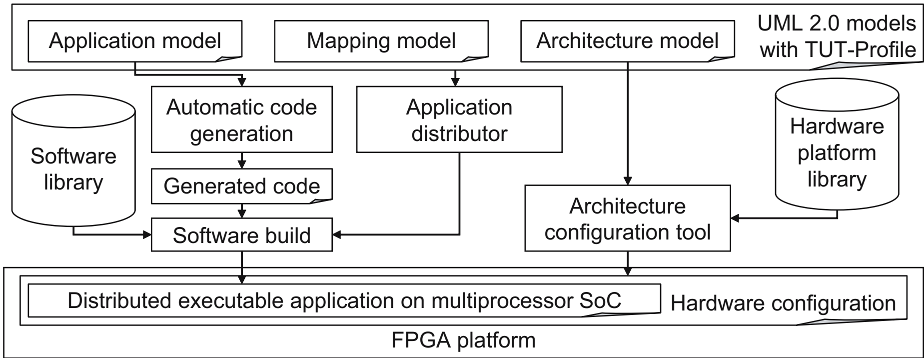

Fig. 2. Implementation flow for UML based multiprocessor systems

3 Automated Implementation Flow from UML to FPGA

The automated implementation flow for UML based multiprocessor systems is presented in Fig. 2. First, UML models for the system are developed. To model hardware and mapping, a UML extension for embedded system design is utilized [12]. The structure and behavior of the application is described in the application model. The application consists of processes described as state-machines. The architecture model defines the processing elements in the hardware, as well as the communication architecture. Applications described in UML are platform independent, and thus application and architecture models can be designed independently. In the mapping model, the application processes are mapped onto the processing elements.

The application model is transformed into executable code using automatic code generation. In the software build the generated code is compiled and linked together with components from a software library, as well as code generated by the application distributor tool. The application distributor is discussed in detail in Chapter 5. As a result, an individual application image for each processor is generated.

Based on the architecture model the hardware configuration is generated automatically by an architecture configuration tool. It generates a model for the top-level architecture where the RTL models from the hardware platform library are instantiated, and performs hardware synthesis.

The application distributor and architecture configuration tool are custom made tools. Telelogic Tau G2 is used for the UML modeling and automatic code generation, and the architecture configuration tool uses Altera Quartus II for the hardware synthesis. For the software build, Nios II GCC toolset is utilized.

4 Case Study: WLAN Terminal

To evaluate the feasibility of our distribution approach in practice, a full featured WLAN terminal was designed in UML and implemented on a prototype platform. TUTWLAN is a proprietary WLAN designed at the Tampere University of Technology (TUT) [13].

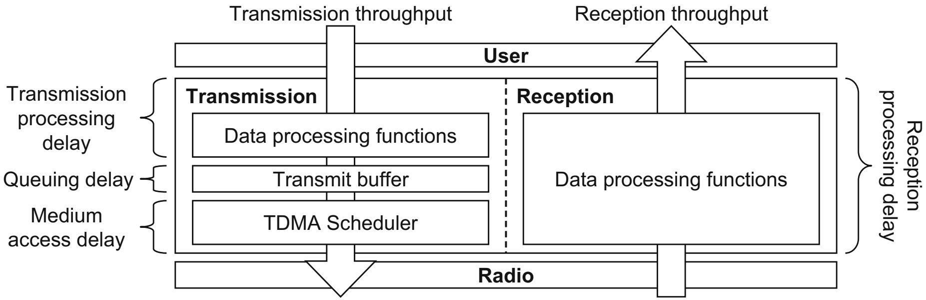

Fig. 3. Key performance parameters of TUTMAC protocol

The central part of TUTWLAN is the TUTMAC protocol, which is a dynamic reservation Time Division Multiple Access (TDMA) based MAC protocol. Several configurations of the TUTMAC protocol have been developed, one of which is presented in this paper. The features of TUTMAC include Quality of Service (QoS) support, data fragmentation, 8-bit CRC for packet headers, and 32-bit CRC and encryption with Advanced Encryption Standard (AES) algorithm for payload data.

The computationally intensive parts of TUTMAC, such as AES encryption, place substantial requirements to the computational capabilities of the hardware platform. Further, the TDMA scheduler has real-time requirements to maintain accurate frame synchronization. Due to its high amount of parallel processing, TUTMAC is capable of reaching its real-time requirements better as a distributed multiprocessor implementation than as a single CPU implementation.

In Fig. 3 the key performance parameters of TUTMAC are illustrated. Processing delay is the time consumed in processing data in transmission or reception. Queuing delay is the time between the arrival of a packet into the transmit buffer and the time the packet is fetched from the buffer. Medium access delay is the time between fetching a packet from the transmit buffer and sending it to the radio. In addition to the actual delays, minimum variation of the delays is a key real-time requirement. Variation of the medium access delay is of special interest, since it affects the accuracy of the frame synchronization.

4.1 TUTWLAN Terminal Configurable Platform

The multiprocessor prototype TUTWLAN terminal is implemented on Altera's Nios II Development Kit, Stratix II Edition. The development board has a Stratix II ES2S60 FPGA and several peripherals such as external memories, serial ports, and an Ethernet controller. An Intersil MACless Prism HW1151-EVAL radio is connected to the prototype expansion headers on the board. The radio is physically IEEE 802.11b compatible but does not implement the standard MAC layer.

The configurable platform consists of multiple Nios II CPUs and custom hardware accelerators, and is presented in detail in [14]. An instance of the platform is shown in Fig. 4. It consists of five CPUs, a radio interface module, and custom hardware accelerators for 32-bit CRC calculation and AES encryption. One of the Nios II CPUs is an I/O CPU, which connects to a wired network via Ethernet.

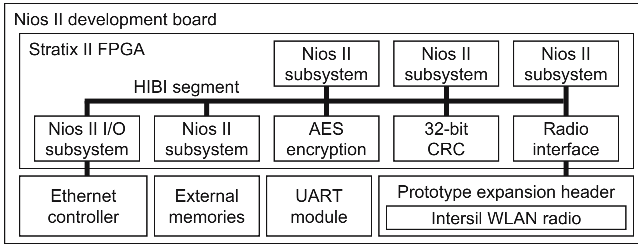

Fig. 4. Platform implementation on the development board

The platform components are connected with a Heterogeneous IP Block Interconnection (HIBI) segment [15], a communication architecture targeted for complex SoC designs. Communication is handled by an Application Programming Interface (API) for HIBI. It offers data transfer services to device drivers for components connected to HIBI, in this case AES, CRC, and WLAN radio.

Each Nios II subsystem is an independent module with local instruction and data memories, and is running a local copy of the eCos RTOS [16]. Using a local copy of an RTOS provides easy scalability, and different operating systems and CPUs can be used in the same architecture. The main benefit of eCos is the real-time kernel providing interrupt and exception handling, multithreading, thread synchronization, and timing mechanisms.

4.2 Protocol UML Model with TUT-Profile

The TUTMAC protocol as well as the hardware architecture and application mapping are modeled in UML 2.0. TUT-Profile defines a set of design practices and stereotypes for embedded system design. The purpose is to enable automated system design using only UML description.

The application components are modeled as classes, and their behavior as statechart diagrams combined with action language. The structure of the application is described with composite structure diagrams, which specify the component instances and their interconnections. The composite structure can be hierarchical, i.e. components can have an internal structure.

Parts of the UML application, e.g. complex algorithms, can be implemented as functions calls. The functions can be implemented either in UML or as external C functions. Further, the C functions can perform the calculation on a hardware accelerator or as a software implementation. If both are available, the decision is based on the process mapping. It is also possible to embed C code directly into the statechart diagrams.

The architecture model is a highly abstracted representation of the actual architecture, described as a composite structure diagram that instantiates the processing elements taken from a platform library. The components in the library have parameterizable UML models, i.e. it is possible to set e.g. different cache sizes for CPUs or different buffer sizes for communication wrappers. Each library component has also an RTL model for the hardware synthesis, and an API for the UML application.

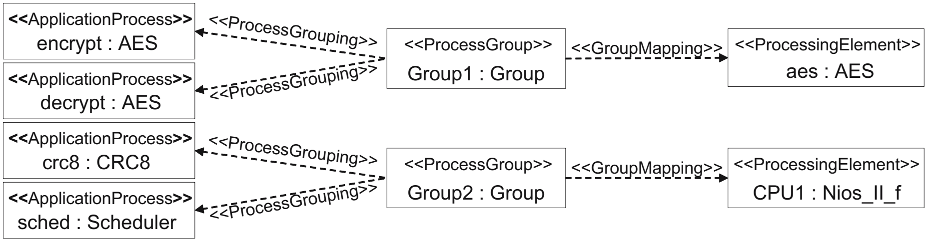

Fig. 5. Mapping of some application processes onto processing elements

Mapping of some application processes onto processing elements is shown in Fig. 5. The mapping is done in two phases: process grouping and group mapping. First, processes are divided into process groups. The grouping can be based on several different criteria, but it is guided by dependencies between processes, such as amount of interaction, different priorities or shared resources. In the second phase, process groups are mapped onto processing elements.

The hardware architecture and mapping in this case have been selected by the designer, but we have developed a sophisticated method for automated architecture exploration for UML based applications [17].

5 Automated Distribution

The automated distribution flow is presented in Fig. 6. The application software consist of code generated by automatic code generation and the application distributor, and software components from a library. The library components contain the platform software, as well as the run-time environment for the distributed execution, consisting of an RTOS API, Inter-Processor Communication (IPC) support, state-machine scheduler, and signal passing functions.

To enable the distributed execution, information about the process mapping needs to be included in the software. The information needed in the distribution is parsed from the UML models with the UML parser tool, which stores the distribution information in a compact XML format. Using this information, the application distributor creates code including a mapping table, defining on which processing element each process group is to be executed, as well as the processes in each group. Information about the signal

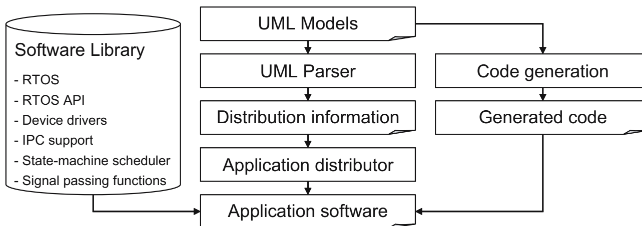

Fig. 6. Automated distribution flow

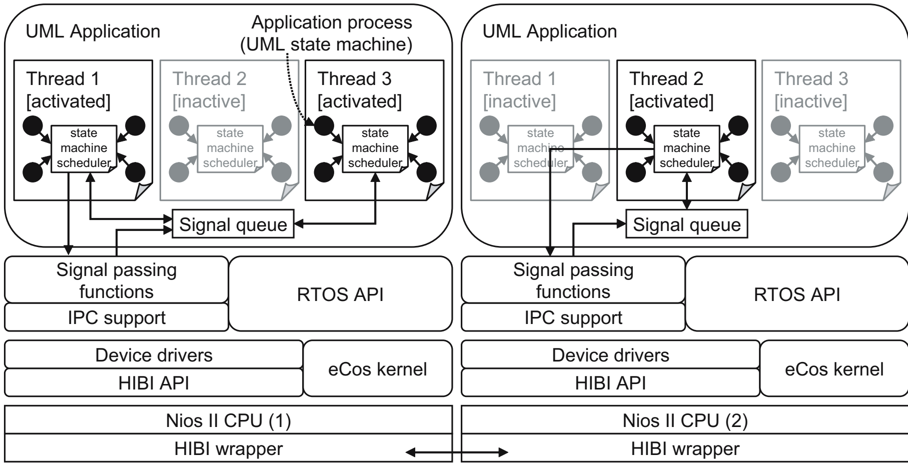

Fig. 7. Run-time environment of a distributed UML application

parameters is also extracted from the generated code to implement the inter-processor signal passing.

Fig. 7 presents the run-time environment for a distributed UML application. In an RTOS, a process group corresponds to a thread, as processes in a group are executed in the same thread. The priority of the groups can be specified in the mapping model, and processes with real-time requirements can be placed in higher priority threads. The execution of processes within a thread is governed by the state machine scheduler. The internal and external signal passing are handled by signal passing functions, which take care that the signal is transmitted to the correct receiver, independent of which CPU the receiver is executed on.

5.1 Scheduling of Application Processes

The same generated code is used for all CPUs in the system, so that each CPU is able to execute all processes of the application. When a CPU starts execution, it checks the mapping table to decide which process groups (threads) it should activate. The signal passing functions take care that signals are delivered to the correct receiver, and the state-machine scheduler does not need the mapping information. All signals are handled in the the order they are received. Inside threads, the state-machine scheduling is non-preemptive, meaning that state transitions cannot be interrupted by other transitions.

The state-machine scheduler is integrated with an underlying operating system by defining an RTOS API, which offers thread creation and synchronization services through a standard interface. Consequently, different operating systems can be used on different CPUs.

5.2 Inter-processor Signal Passing for UML Processes

The signal passing functions need services to transfer the UML signals between different CPUs. The IPC support provides these services by negotiating the data transfers

and handling possible data fragmentation. On the TUTWLAN platform, it uses the HIBI API for the data transmissions.

State transitions occur when a UML process sends a signal which triggers a transition in another process, or when a timer expires. Signals are comprised of a standard header and payload data. Header includes information about the signal sender, receiver and priority. Payload consists of zero or more parameters, which may have different data types including integer values, strings, and arrays.

The signal passing at run-time is performed using two signal queues: one for signals passed inside the same thread and other for signals from other threads. Processes within a thread share common signal queues. When a signal is received, it is placed to the corresponding queue.

Our run-time environment extends this basic functionality by enabling the communication between CPUs. When the state-machine scheduler detects that a signal is sent to a process which resides on a different CPU, the signal passing functions transmit the signal to the signal queue on the receiving CPU. The mapping table is used to determine the target CPU. The signal passing functions use the code generated by the application distributor to find information about the parameters of the signal. Based on this information, the signal header and parameters are copied and sent to the receiver using the IPC support.

When a signal is received, the IPC support passes the signal to the signal passing functions. A UML signal is reassembled and added to the external signal queue. The state-machine scheduler fetches the signal from the queue and passes it to the receiving process. From the point of view of a UML process, IPC is transparent; the reception of a signal is exactly the same for signals from all processes in the system, regardless of which CPU or thread the sending process is executed on.

5.3 Dynamic Mapping

The context of a UML state machine is completely defined by its current state and the values of its internal variables. Since threads have a common memory space, and the variables are stored as global data structures, it is possible to change the mapping of application processes between threads at run-time by updating the mapping table and moving existing signals to correct queues.

Further, since all CPUs use the same generated code, it is possible to re-map processes between processing elements at run-time. This operation is somewhat more complicated, since it involves transferring the state-machine variables and signals between CPUs. This issue is beyond the scope of this paper and will be considered in detail in future publications.

6 Performance Measurements

A distributed multiprocessor implementation of an application is expected to affect the performance in terms of reduced execution time and variance. However, the distribution itself causes overhead both in memory requirements and in execution time. The experimental measurements were performed to evaluate the absolute amount of overhead as well as the total effect on performance.

Table 1. TUTMAC static memory requirements for a single CPU

Software component	Code (bytes)	Data (bytes)	Total (bytes)
Generated code	21 748	1 916	23 664
State-machine scheduler	26 064	13 137	39 201
External functions	22 600	33 889	56 489
Signal passing functions	7 064	10 764	17 828
HIBI API	3 552	36 100	39 652
IPC support	3 304	14 084	17 388
Device drivers	2 940	212	3 152
eCos	49 576	5 818	55 394
Total software	136 848	115 920	252 768

6.1 Resource Usage

The static memory requirements for a single CPU are given in Table 1. For each software component the code and data memory requirements are shown. The total memory requirements for a complete multiprocessor system can be evaluated by multiplying the requirement of a single CPU with the number of CPUs in the system.

In addition to the static requirements, TUTMAC uses dynamic memory for signaling between UML processes, and to buffer outgoing packets. The signaling requires approximately 4-5 kilobytes of memory, and depending on the size of the transmit buffer, the total dynamic memory usage is 50-100 kilobytes. Dynamic memory usage is independent on the number of CPUs. Altogether, a TUTWLAN terminal with four CPU's requires approximately 1.1 megabytes of memory.

The Stratix II FPGA used in the prototype implementation has 60,440 equivalent 4-input look-up tables and 2,544,192 bits of on-chip RAM memory. A TUTWLAN terminal configuration consisting of five Nios II CPUs and AES, CRC, and radio interface modules requires 69% of the logic elements and 36% of the on-chip memory.

6.2 Performance Evaluation

To evaluate the performance overhead caused by distribution, delays for signals passed between processes were measured in three different scenarios: two communicating processes on the same thread, on different threads on the same CPU, and on different CPUs. The measured delays are shown in in Fig. 8. In each case the size of the signal payload affects the delay, since the data is copied. On different CPUs, the delay is also increased because of the IPC. If a data type passed as a pointer is used, then the delay would be independent of the payload size between processes on the same CPU, but between CPUs the data is always copied because of local memories.

The delay between different CPUs is 2.5-3 times larger compared to the delay between different threads, depending on the payload size. However, measurements with the TUTMAC protocol show an increase in the total performance of an application by distribution.

The measurements were performed on an architecture configuration consisting of four CPUs and the AES hardware accelerator. Protocol functionality was divided to

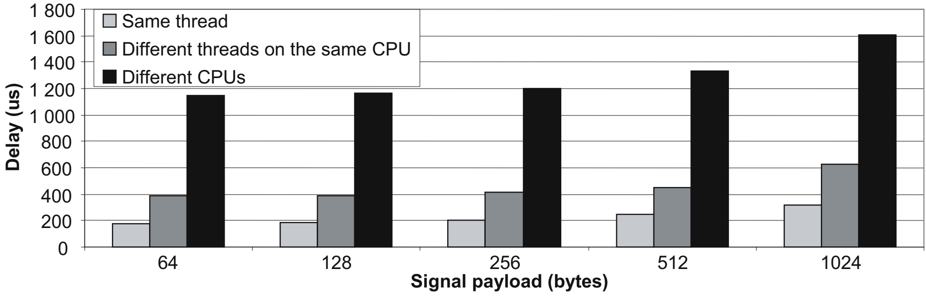

Fig. 8. Delays for UML signal passing in three different scenarios

Table 2. Mappings used in the performance measurements

Process group	Mapping 1 CPU #	Mapping 2 CPU #	Mapping 3 CPU #	Mapping 4 CPU #	Mapping 5 CPU #
Control	1	1	1	1	1
Data processing (reception)	1	2	2	2	2
Data processing (transmission)	1	1	3	3	3
AES decryption	1	2	2	4	Hardware acc.
AES encryption	1	1	3	4	Hardware acc.

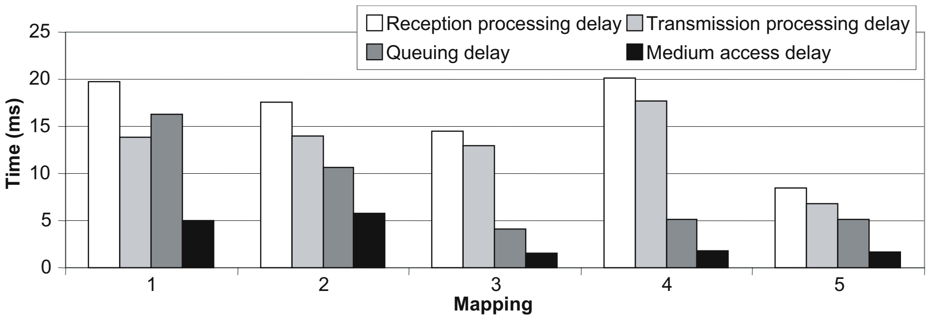

Fig. 9. Measured delays for TUTMAC protocol with different mappings

five process groups, and TUTMAC was executed using different mappings which distributed the groups to different processing elements according to Table 2. The control group includes the TDMA scheduler, 8-bit CRC, radio interface, frame buffering and protocol management. The data processing groups include 32-bit CRC, fragmentation and user interface processes for transmission and reception, respectively. The last two groups include the AES encryption and decryption.

The measured protocol delays are presented in Fig. 9. The results show that the distribution has a significant effect on the execution times. The most notable effect is

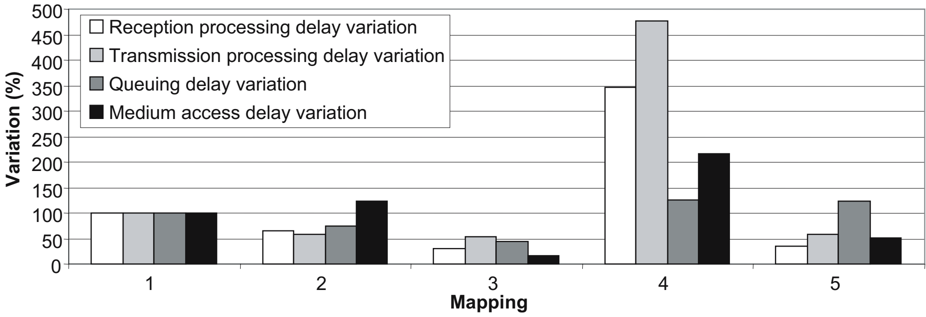

Fig. 10. Measured delay variations for TUTMAC protocol with different mappings

the reduction of the queuing delay, which with three CPUs is reduced to just 25.4% of the original delay with a single CPU. In the first two mappings, the TDMA scheduler is unable to meet all of its real-time requirements and some TDMA data slots are missed. Thus, frames cannot be transmitted at full speed, which increases the queuing delay.

With three CPUs and the AES encryption module, the real-time requirements are fulfilled and a 50-70% speed-up is achieved for all delays when compared to the single CPU mapping. Using a fourth CPU for the AES calculation proved to be inefficient in terms of reducing the delays, since the AES calculation is the most time consuming function of the protocol and thus it is not efficient for transmission and reception to share a common CPU for it. However, in the fifth mapping the hardware accelerator can perform the AES calculation fast enough to reduce both processing delays.

Changes in the delay variations compared to the single CPU mapping are shown in Fig. 10. With three CPUs, the medium access delay variation is reduced by 84.0%. Further, all other delay variations are also reduced 45-70%. These improve the accuracy of TDMA scheduling and the robustness of the system.

7 Conclusions

This paper presented automated distribution of applications modeled in UML to a multiprocessor SoC. With the aid of design automation and modeling of hardware and mapping in UML, the distribution is fully automated and straightforward.

The case study showed that the approach is feasible, and the measurements performed on the prototype platform showed significant improvement in the protocol performance. The future work will include measuring the efficiency overhead incurred by high-level UML design compared to other implementation approaches. The state-machine scheduler and IPC should be optimized to reduce the distribution overhead. Further, possibilities of the dynamic process re-mapping in power management will be studied.

References

1. Object Management Group (OMG): UML 2.0 Superstructure Specification (Version 2.0). (2004)
2. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. In: Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems. (1996)
3. Kaiserswerth, M.: The Parallel Protocol Engine. IEEE/ACM Transactions on Networking **1** (1993) 650–663
4. Object Management Group (OMG): The Common Object Request Broker Specification (Version 3.0). (2004)
5. Schmidt, D.C., Kuhns, F.: An overview of the real-time corba specification. Computer **33** (2000) 56–63
6. Brinkschulte, U., Ungerer, T., Bechina, A., Picioroaga, F., Schneider, E., Kreuzinger, J., Pfeffer, M.: A microkernel middleware architecture for distributed embedded real-time systems. In: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems. (2001) 218–226
7. Gill, C., Subramonian, V., Parsons, J., Huang, H.M., Torri, S., Niehaus, D., Stuart, D.: ORB middleware evolution for networked embedded systems. In: Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems. (2003) 169–176
8. Chen, R., Sgroi, M., Lavagno, L., Martin, G., Sangiovanni-Vincentelli, A., Rabaey, J.: UML and platform-based design. In: UML for Real: Design of embedded Real-time Systems. Kluwer Academic Publishers (2003) 107–126
9. Martin, G., Lavagno, L., Louis-Guerin, J.: Embedded UML: A merger of real-time UML and co-design. In: Proceedings of the Ninth International Symposium on Hardware/Software Codesign. (2001) 23–28
10. Object Management Group (OMG): UML Profile for CORBA Specification (Version 1.0). (2002)
11. Born, M., Holz, E., Kath, O.: A method for the design and development of distributed applications using UML. In: Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems. (2000) 253–264
12. Kukkala, P., Riihimäki, J., Hännikäinen, M., Hämäläinen, T.D., Kronlöf, K.: UML 2.0 profile for embedded system design. In: Proceedings of the Design, Automation and Test in Europe. Volume 2. (2005) 710–715
13. Hännikäinen, M., Lavikko, T., Kukkala, P., Hämäläinen, T.D.: TUTWLAN - QoS supporting wireless network. Telecommunication Systems - Modelling, Analysis, Design and Management **23** (2003) 297–333
14. Arpinen, T., Kukkala, P., Salminen, E., Hännikäinen, M., Hämäläinen, T.D.: Configurable multiprocessor platform with RTOS for distributed execution of UML 2.0 designed applications. In: Proceedings of the Design, Automation and Test in Europe. (2006)
15. Salminen, E., Lahtinen, V., Kangas, T., Riihimäki, J., Kuusilinna, K., Hämäläinen, T.D.: HIBI v.2 communication network for system-on-chip. In: Proceedings of the International Workshop on Systems, Architectures, Modeling and Simulation. (2004) 413–422
16. Massa, A.J.: Embedded Software Development with eCos. Prentice Hall PTR (2002)
17. Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., Hämäläinen, T.D., Riihimäki, J., Kuusilinna, K.: UML-based multi-processor SoC design framework. Accepted on ACM Transactions on Embedded Computing Systems (2006)