A FRAMEWORK FOR GENERIC HW/SW COMMUNICATION USING REMOTE METHOD INVOCATION

Philipp A. Hartmann, Kim Grüttner

OFFIS Institute for Information Technology
Oldenburg, Germany
{hartmann,gruettner}@offis.de

Philipp Ittershagen, Achim Rettberg

Carl von Ossietzky University
Oldenburg, Germany
{ittershagen,rettberg}@uni-oldenburg.de

ABSTRACT

Implementation of communication between different tasks of a concurrent embedded system is a challenging problem. The aim of our work is to support the refinement and relocation of tasks onto different execution units, such as processors running different operating system or even dedicated hardware. For this purpose communication should be transparent and as independent as possible from the underlying middleware or embedded operating system. Moreover, communication should also be transparent accros the HW/SW boundary.

In this work we present a generic framework for seamless communication of (software) tasks with shared resources, called *Shared Objects*. Communication is implemented using a method-based interface realizing a *Remote Method Invocation* (RMI) protocol. Our shared communication resources can either be implemented as dedicated hardware, as shared memory or local tasks. The presented framework is a first step towards the unification of shared resource access based on embedded Linux. The effectiveness of our approach is be evaluated with different task mappings and shared resource access implementation styles.

Index Terms— HW/SW communication, RMI, shared resources, driver synthesis

1. INTRODUCTION

The design of complex embedded systems includes the partitioning into hardware and software blocks. Implementing system functionality in dedicated hardware instead of software usually results in a boosted system performance, but at higher costs, reduced flexibility and maintainability. The conclusion could be to use software-only systems to get rid of these nasty drawbacks. But for some systems a software-only solution is still infeasible due to limited processing power and the fulfillment of a certain power budget.

For the exploration of different HW/SW partitioning a flexible and transparent communication infrastructure is an important requirement. A design methodology should assist the designer during this partitioning and implementation on different target architecture to meet cost and time to market constraints [1]. For validating the system behavior under different HW/SW partitioning and mapping alternative the overall system shall be executable at any time during the design process. This execution, also called simulation, enables to derive first information to validate design decisions and drive the refinement process. A homogeneous design environment where hardware and software blocks are represented in the same description language eases a joint simulation and system evaluation. Moreover, it enables easy and rapid changes in the system partitioning.

When splitting up a system into hardware and software blocks the execution times and also the communication times between these blocks need to be analyzed. This becomes a crucial task when the system needs to meet certain timing requirements. Different HW/SW partitioning and bindings of these subsystems to different hard and software components define these execution and communication times. For a flexible change of these bindings we propose a transparent communication mechanism and synthesis support for communication across the HW/SW boundary.

The refinement and evaluation of inter-process communication is a central part of the OSSS design methodology. To get from an executable system specification in OSSS to an implementation on a specific hardware platform, the HW/SW communication and synthesis is an essential issue. In this work we present our driver framework *rmi4linux* that provides a run-time environment for OSSS on Linux based embedded platforms. For this purpose we will start with a short introduction of the OSSS design methodology in Section 2. In Section 4 the main requirements for the driver framework are presented. Driven by these requirements in Section 5 the implementation of the framework, the HW/SW interface between *rmi4linux* and an OSSS *Shared Object*, and support for *Software Shared Objects* are described. In Section 6 a first evaluation of the proposed framework is presented.

2. PREVIOUS WORK – OSSS

The presented work is based on the OSSS (Oldenburg System Synthesis Subset)[2] design methodology. OSSS sup-

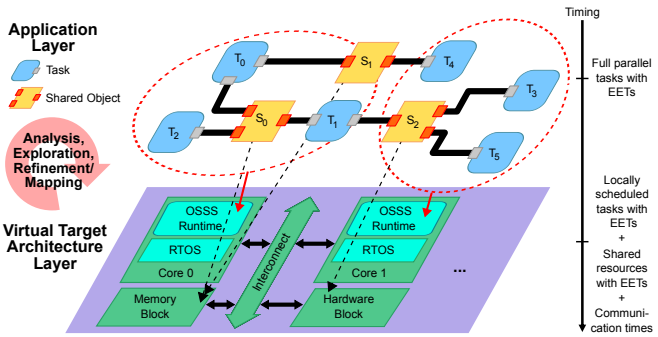


Fig. 1. Overview of the OSSS design methodology [4].

ports the designer in modeling complex embedded systems through a homogeneous description and refinement methodology for hardware and software components.

OSSS is an extension of the modeling and simulation environment SystemC [3]. In addition to SystemC it offers dedicated and synthesizable primitives for *Hardware Modules*, *Software-Tasks*, and *Shared Objects* as main communication primitve. *Shared Objects* enable an object-oriented description of shared resources and provide a method-based interface. Parallel running hardware modules and software tasks (both called *Clients*) are synchronized transparently using user-defined *Shared Objects*.

The initial system in OSSS is modeled in the so called *Application Layer* (Figure 1). The system consists of parallel tasks which are communication and synchronized through *method calls* on *Shared Objects*. Required and Estimated Execution Times (RET, EET) can be annotated to the initial tasks for enabling first design decisions and timing validation through simulation.

The *Shared Object* modeling primitive enables an object-oriented design of parallel applications. A Shared Object provides a user-defined method interface to its accessing clients with extended synchronization features: Method calls to a *Shared Object* are executed in a critical section and can be linked with logical preconditions on the internal state of the object, so-called *Guards*. Based on the internal state of the object, method calls can be blocked until the guard condition evaluates to `true`. When a guard condition is `false` it can only be changed to `true` by modification of the object's internal state via method calls of non-blocked client processes.

For the exploration of different architecture alternatives, OSSS provides the *Virtual Target Architecture* (VTA) *Layer*. This layer represents abstract and virtual components of the target platform (software processors, memories, dedicated hardware, and communication infrastructure). Tasks and objects from the Application Layer are mapped to specific components in the VTA layer. Tasks and also Shared Objects can be mapped to dedicated hardware blocks as well as software processors and memories [4]. The resulting model after this mapping is also executable and adds more precise timing information regarding scheduling of multiple tasks on a single software processor, inter- and intra-processor communication delays and congestions due to shared resource access conflicts. In [5] the simulation of an abstract operating system, including the resulting scheduling artifacts is described.

For communication with remote Shared Objects the *Remote Method Invocation* (RMI) technique for mapping abstract method calls to a platform and system specific communication protocol is applied [6] transparently, without requiring changes to the Application Layer input model. In the case of a dedicated hardware implementation Shared Objects have their own hardware scheduler to serialize incoming method requests. This scheduler performs arbitration among all pending client requests to enable unblocked method calls leading to internal state changes for resolving blocking conditions of other methods.

For the transformation of the VTA layer model into an implementation on the chosen target platform, all Tasks and Shared Objects which have been mapped to dedicated hardware are synthesized to RT level VHDL code using the *Fossy* (Functional Oldenburg System SYnthezizer) synthesis tool. For Tasks and Shared Objects mapped to software we provide a run-time layer which maps the semantics of *Software Tasks* and *Shared Objects* on primitives of the underlying operating system. In this paper we present our generic implementation approach of RMI communication based on embedded Linux as the chosen run-time layer for software synthesis. This provides the seamless integration of the (software) caller's site of the RMI protocol implementation, again automatically derived from the input model.

To represent different access implementations to shared resources, different communication scenarios between tasks and *Shared Objects* need to be considered [4]. This includes communication with remote hardware objects, remote software objects, and local software objects using "regular" critical sections based on existing operating system primitives.

3. RELATED WORK

In literature several approaches for automatic generation of drivers for HW/SW communication can be found [7, 8]. In some early work [9] an automatic approach for the generation of application specific heterogeneous multi processor architectures has been proposed. The intention of this work, to automate error-prone manual implementation of communication channels between hardware and software, is the same as for our approach. A possibility for the automatic generation of Linux drivers has been studied in [10]. This approach proposes a generic interface as well as an abstraction of the driver development itself. For the comparison of our work we will focus on SystemC-based approaches with the main goal to automatically derive the implementation of HW/SW communication for embedded SoCs.

In [11] a technique for automatic software driver generation for HW/SW communication based on SpecC is presented. This approach is comparable to ours. In difference to Shared Objects the SpecC approach uses simpler channel models which allow the implementation of a critical section through mutexes. The scheduling of communication in SpecC is modeled explicitly during the computational refinement, while Shared Objects have built-in scheduling capabilities that allow the implementation of dynamic scheduling in hardware and software.

In [12] a SystemC-based software modeling approach on top of a POSIX compatible simulation library is proposed. The designer can perform timed SW simulations while considering the impact of a POSIX compatible RTOS. This approach also performs automatic code generation of the supported operating systems. The main drawback of this approach is the missing differentiation between communication of tasks on different operating systems and the missing support of communication with dedicated hardware resources.

The work in [13] presents an approach for the systematic refinement of distributed embedded systems. Using SystemC the overall system is modeled, including a step-wise refinement to a virtual prototype with clock cycle accuracy. Along with this refinement process SW code with RTOS primitives and drivers for a packet-based network communication (CAN) are generated. The main difference to our approach is that only packed-based network communication drivers are considered. Our work focuses on SoC communication using shared memory architecture with a special emphasis on shared resource access arbitration.

4. REMOTE METHOD INVOCATION – REQUIREMENTS

To enable the seamless refinement of specific applications on top of a predefined, yet possibly customized platform, an appropriate hardware/software interface is needed. Software Tasks should not need to be heavily modified, when different mappings of shared resources to alternative implementations (dedicated hardware, shared memory, core/OS-local primitives) are chosen. In this work, this unified interface is based on *Remote Method Invocation* (RMI). The RMI protocol across the HW/SW boundary is characterized by four different states, that are sequentially processed [6]:

1. Serialization of the arguments for the requested method.
2. Notification of the Shared Object, sending the unique Client ID, and the requested method ID to the hardware Shared Object. The Shared Object's local scheduler arbitrates concurrent requests and grants the access eventually.
3. Sending of the arguments to the Shared Object, deserialization and execution of the requested functionality within the Shared Object.

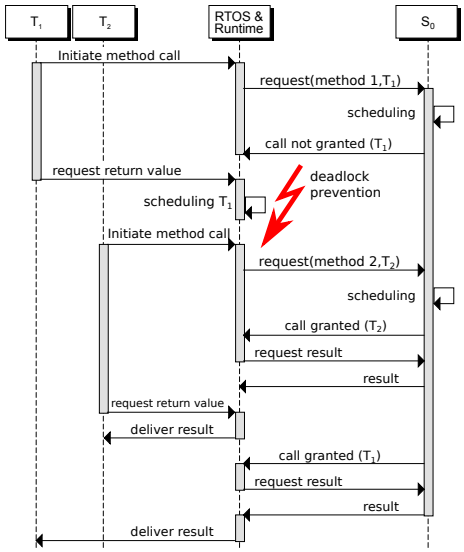


Fig. 2. Example sequence of two Software Tasks accessing a Shared Object, following the configuration of tasks and objects in Figure 1.

4. An optional return value is sent back to the client (using the same, but reverse serialization mechanism)

Since multiple clients of a single Shared Object may be sharing the same processor core, the implementation of the RMI protocol requires additional runtime support by the OS or a specific driver as shown in Figure 2. The reason for this extended synchronization needs is caused by the multi-level locking of the Shared Object in terms of (a) mutually exclusive access of concurrent requests and (b) the potential blocking due to the guard conditions.

If the request of a local client task is blocked by a non-granted guard condition, it still needs to release the (local) lock of the Shared Object, to avoid a (local) deadlock: A second task running on the same core as the currently blocked task would be locked out of this Shared Object, potentially never modifying the state of the Shared Object to change the guard condition currently blocking the first task.

In addition to the support of this more complex synchronization requirements, a unified interface to different flavors of Shared Object implementations increases the decoupling of computation and communication and enables seamless refinement. This includes the so-called *Software Shared Objects*, where the method execution is performed within the context of the calling process directly. They can either be built on top of OS synchronization primitives (in case of all clients being managed by the same OS instance), or in terms of explicitly shared memory used from multiple cores. In the latter case, the locking mechanism needs to be implemented explicitly. Summarizing, the requirements for the generic driver framework are the following:

- Access from multiple clients (running on the same or different cores) to multiple Shared Objects of different flavors shall be possible.
- A common, transparent abstraction from the actual binding/mapping of the Shared Object (→ Hardware, shared memory, local memory) shall be available to the client.
- To increase the decoupling and enable further parallelization, the communication between clients and (Hardware) Shared Objects should support an asynchronous protocol.
- The implementation should separate platform/mapping properties from the application and library code, to enable seamless refinement and adjustments in that regard.

Our prototypical implementation *rmi4linux* of such a framework for generic HW/SW communication based on OSSS RMI is based on the widely available open-source Linux operating system.

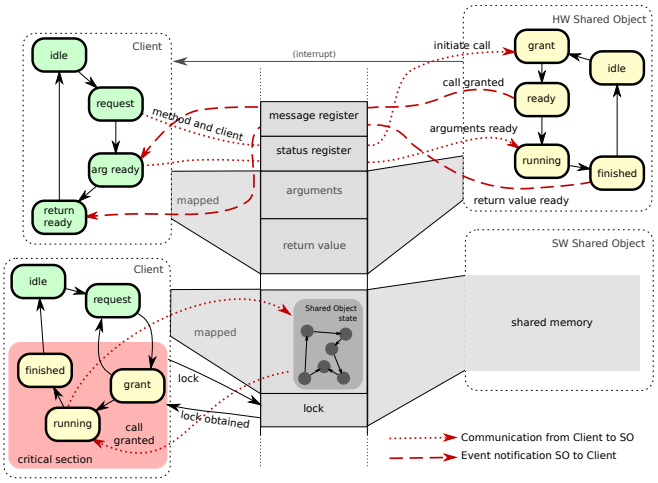


Fig. 3. Interaction and memory layout of Client and Shared Object.

5. IMPLEMENTATION BASED ON LINUX

The *rmi4linux* driver frameworks consists of a Linux kernel module and an accompanying user-space library for increased usability of the low-level driver from clients, which are implemented as regular user processes. The platform-specific information, like the sets of defined methods or physical address ranges are externally stored within a so-called *device tree* [14]. This enables the separation of platform-dependent information from the actual implementation and increases reuse and flexibility.

The communication between a client and the driver framework is based on `ioctl` calls signaling the beginning and end of an RMI operation. The transfer of arguments and result values is done via dedicated, mapped address ranges. Upon initialization of a (new) client, such a memory is prepared and directly and exclusively mapped to the address space of the client. This avoids additional copying from user-space to kernel-space, since the client has direct access to this memory.

The RMI protocol requires a synchronization across the HW/SW boundary. For instance, a method cannot be invoked until the guard condition for this method is fulfilled, in which case the driver needs to wait. Only when the explicit grant is received, the arguments can be sent to the Shared Object and the call can be performed. Similarly, the framework needs to wait for the end of the Shared Object computation, before the results can be read.

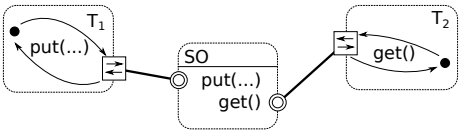
In the implementation of these two synchronization phases during an RMI (waiting for a grant, waiting for the results), two variants are supported: (a) polling of the corresponding control registers; (b) an interrupt-driven approach. The first variant can be used without explicit hardware support in the "software-only" mapping as well, while the second alternative requires dedicated Hardware Shared Objects.

To enable a remote method call, the driver needs information about the internal state of the Shared Object. Especially the notification of a possible start and the completion (and available results) is important here. The different mappings to either hardware or a shared memory differ in that regard. Since several RMI operations may be processed concurrently, the initiating client needs to be known to distinguish these operations. Each client can only issue a single request to a Shared Object at a time. Therefore, each client-specific status register in the interface contains the currently requested method ID as well as the current state of the execution (idle, grant, ready, running, finished). This enables the communication between client and Shared Object and notification of pending argument/result transfers between the two, as depicted in Figure 3.

Each Shared Object has a single, additional *message register* used to send messages to a client. The client needs to react upon reception of a grant for a pending request (and send the arguments), as well as when the execution of a request has finished (to receive the results). These particular events are notified by the Shared Object via the message register. The client (or rather the driver framework) handles these messages during interrupt-handling or through polling.

While the communication with a Hardware Shared Object is handled via memory-mapped IO based on a dedicated register interface, Figure 3 shows the alternative mapping on top of a plain shared memory region in case of inter-core Software Shared Objects. In this scenario, the methods are executed within the context of the requesting client task. In addition to the internal state of the Shared Object, an additional `lock` flag is mapped to the shared memory, used to enable the exclusive access of multiple clients (critical section).

To execute a method of such a Software Shared Object,



(a) Task and Shared Object interaction

Exp.	Event indication	CPUs		Shared Object	
		CPU_1	CPU_2	Hardware	Memory
1	interrupts	T_1,T_2	-	SO	
2	interrupts	T_1	T_2	SO	
3	polling	T_1	T_2	SO	
4	polling	T_1	T_2		SO

(b) Configuration of different experiments

Fig. 4. Experimental setup

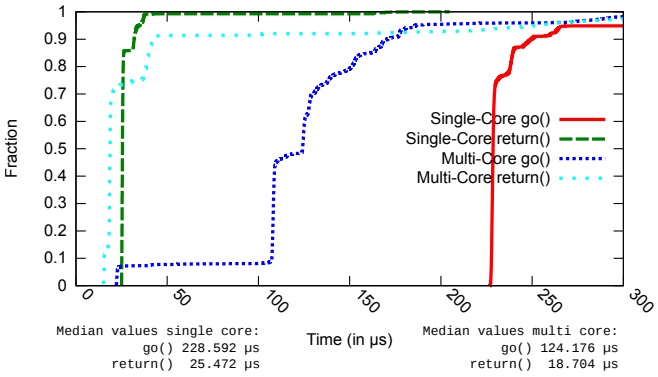
the client has to gain exclusive access to the object's state by obtaining the lock. Secondly, the grant conditions of the method need to be evaluated. When granted, the requested method can be executed. Afterwards, and in case of a non-granted guard condition, the lock is released again, as shown in Figure 3. The explicit implementation of the lock mechanism is required to enable synchronization across core and OS boundaries.

The different low-level interfaces (HW with or without interrupt, shared memory, local memory) is hidden from the client behind the generic user-space runtime library, enabling transparent access to all the different implementation flavours of a Shared Object. Platform- and mapping-specific configuration is handled separately in the *device tree* [14].

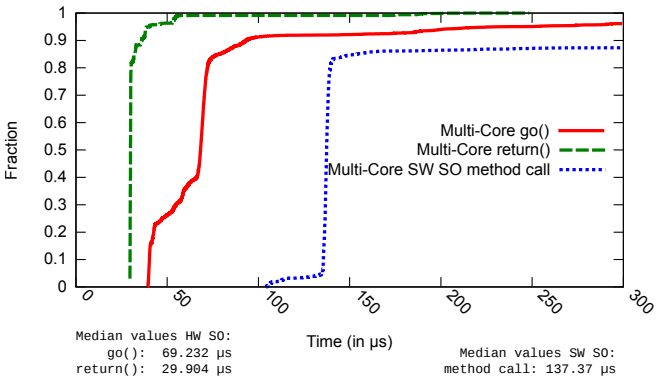
6. EVALUATION

To evaluate the *rmi4linux* framework, a set of experiments running on an embedded multi-processor design have been performed. The system implementation is based on a Xilinx ML410 FPGA board, providing a Virtex-4 FPGA with two built-in PowerPC cores. The system is running at 100 MHz and incorporates two separate interrupt controllers for the different cores to enable the asynchronous protocol notification with the (hardware) Shared Objects.

Based on the basic producer/consumer example shown in Figure 4(a), four different experiments with different mappings and implementations have been conducted. In each experiment, two tasks are accessing a single Shared Object, implementing a FIFO buffer with a capacity of a single element. The tasks are calling the `put`, and `get` functions of the FIFO object as fast as possible in an infinite loop. The reduced size of the FIFO buffer has been chosen deliberately to demonstrate the contention on this scarce resource between the two different clients, especially due to blocking guard conditions (empty/full FIFO). This enables the analysis of the different



(a) Cumulative distribution of the duration of a method call and the request for a result on a HW Shared Object using interrupts.



(b) The duration of a HW and SW Shared Object method call when using polling instead of interrupts.

Fig. 5. Evaluation results: method call/protocol cost

implementation variants on the blocking times of the tasks.

The FIFO Shared Object has been mapped to either a dedicated hardware block (Exp. 1–3), or to a shared memory region (Exp. 4). The tasks are either mapped to the same (Exp. 1), or to different PowerPC cores, each of which is running its own instance of the Linux OS (Exp. 2–4). The event notification required for the RMI protocol is either performed in terms of interrupts (Exp. 1–2) or by polling the corresponding memory (mapped) locations (Exp. 3–4). Table 4(b) gives a summary of the different configurations. The missing combinations are either not possible or not feasible: (i) plain shared memory has no interrupt support, and (ii) in a single-core scenario, a guard implementation based on polling would result in a deadlock or at least a huge delay (when using time-sliced scheduling) until the blocked task releases the CPU to give the other task the opportunity to release the guard.

For the measurements of the different execution times, the *function tracer* [15] of the Linux kernel has been used. Since the activation of such an online measurement impacts the function execution times itself, only the functions to asynchronously initiate the RMI (`go`) and the potentially blocking wait for the result (`return`) have been traced.

The timing behavior for accessing an interrupt-enabled hardware Shared Object is shown in Figure 5(a). It can be seen, that in case of Exp. 2 the median time to initiate a function call is roughly half of the delay required for the request in Exp. 1. The cause for this is the distributed load of the interrupt-handling on the two cores: In case of Exp. 2, every core has to handle only half of the interrupts. And due to the very short computation delay in the hardware itself, the interrupts usually arrive already during the initiation of the function call, and therefore delaying its completion locally. On the other hand, the early completion of the hardware execution leads to non-blocking access to the method's return values. This can be seen in terms of significantly shorter execution times for the `return` handling in Figure 5(a).

Consequently, the polling-based implementation, both with a dedicated hardware Shared Object and with a pure memory-based implementation lead to better performance, as shown in Figure 5(b). In case of the pure software implementation, the graph shows the overall execution time for both request and return path, since both parts cannot be separated reasonably well. Still the overall execution time is comparable to the hardware-supported implementation. The main reason for the benefits of polling in case of a FIFO communication lies in the small computational delays within the Shared Object, leading to very short locking periods, where the cost of context switches and interrupt-handling outweighs the short periods of busy waiting.

Due to the standard Linux kernel scheduler, there are some quite expensive corner cases visible, due to missed time-slices and interfering background behavior. We are currently working on the integration of the real-time enhancements for Linux to improve the predictability in that area.

7. CONCLUSION

In this work, the driver framework *rmi4linux* has been presented, that provides a unified interface to software tasks accessing OSSS *Shared Objects* for multi-processor systems-on-chip running (multiple instances of) the Linux OS. The selection of a particular, supported implementation variant of the Shared Object (dedicated hardware, shared memory, OS-local) is hidden from the accessing task behind a common abstraction. This enables late changes to the architectural mapping without costly adoption of the application model.

In Section 6, different mapping alternatives have been analyzed for a simple, illustrative example. It has been shown, that the synchronization cost due to the blocking on the scarce resource significantly depends on the chosen mapping. This is another compelling reason for early simulation models, which can be based on OSSS as well [2, 5].

We're currently working on a trade-off analysis to explore the break-even point between the interrupt-based event notification overhead compared to the blocking time of the Shared Object in case of polling-based implementation. For run-time reconfigurable HW accelerator blocks with long reconfiguration delays, such an approach may still be beneficial. Secondly, a (cycle accurate) virtual-platform based execution time measurement is under development as well. In general, the physical execution times can be back-annotated to the simulation model to speed-up the design space exploration at a higher abstraction level.

8. REFERENCES

[1] D. Gajski, J. Zhu, and R. Dömer, *Hardware-Software Co-Synthesis – Principles and Practice: Essential Issues in Codesign*, Kluwer Academic Publishers, 1997.

[2] OFFIS Institute for Information Technology, "OSSS – A Library for Synthesisable System Level Models in SystemC," Tech. Rep., Oldenburg, Germany, 2008.

[3] IEEE P1666™-2005, *Standard SystemC Language Reference Manual*, 2005.

[4] P. A. Hartmann, K. Grüttner, A. Rettberg, and I. Podolski, "Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC," in *DIPES'10: IFIP Conference on Distributed and Parallel Embedded Systems*, Sept. 2010.

[5] P. A. Hartmann, P. Reinkemeier, H. Kleen, and W. Nebel, "Modeling of Embedded Software Multitasking in SystemC/OSSS," in *Languages for Embedded Systems and their Applications*, vol. 36 of *Lecture Notes in Electrical Engineering*, pp. 213–226. Springer Netherlands, 2009.

[6] K. Grüttner, C. Grabbe, F. Oppenheimer, and W. Nebel, "Object Oriented Design and Synthesis of Communication in HW/SW Systems with OSSS," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), Hokkaido, Japan*, Oct. 2007.

[7] S. Shibata, S. Honda, H. Tomiyama, and H. Takada, "Advanced SystemBuilder: A Tool Set for Multiprocessor Design Space Exploration," in *Intl. SoC Design Conference (ISOCC'2010)*, Nov. 2010, pp. 79–82.

[8] M. Diaby, M. Tuna, J.-L. Desbarbieux, and F. Wajsburt, "High Level Synthesis Methodology from C to FPGA Used for a Network Protocol Communication," in *Proc. of the* 15^{th} *IEEE Intl. Workshop on Rapid System Prototyping*, Washington, DC, USA, 2004, pp. 103–108, IEEE Computer Society.

[9] S. Vercauteren, B. Lin, and H. De Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications," in *Design Automation Conference (DAC'1996)*, 1996, pp. 521–526.

[10] T. Katayama, K. Saisho, and A. Fukuda, "Prototype of the device driver generation system for UNIX-like operating systems," in *Intl. Symposium on Principles of Software Evolution (ISPSE'2000)*, 2000, pp. 302–310.

[11] H. Yu, R. Dömer, and D. Gajski, "Software and driver synthesis from transaction level models," in *From Specification to Embedded Systems Application*, vol. 184 of *IFIP International Federation for Information Processing*, pp. 65–76. 2005.

[12] F. Herrera, H. Posadas, P. Sánchez, and E. Villar, "Systematic Embedded Software Generation from SystemC," in *in Proc. of DATE*, 2003.

[13] W. Rosenstiel M. Krause, O. Bringmann, "Communication Refinement and Target Software Generation using SystemC," in *Proc. of MBMV'2006*, Dresden, 2006.

[14] D. Gibson and B. Herrenschmidt, "Device trees everywhere," Dunedin, New Zealand, Feb. 2006, OzLabs, IBM Linux Technology Center.

[15] T. Bird, "Measuring function duration with ftrace," in *Proc. of the Linux Symposium*, Montreal, Canada, July 2009, pp. 47–54.