A Radical Approach to Network-on-Chip Operating Systems

Michael Engel and Olaf Spinczyk
Technische Universität Dortmund
Embedded System Software Group
{michael.engel,olaf.spinczyk}@tu-dortmund.de

Abstract

Operating systems were created to provide multiple tasks with access to scarce hardware resources like CPU, memory, or storage. Modern programmable hardware, however, may contain a large number of processing units on one chip working independently as system-on-chip (SoC).

It is feasible to build a system that provides each task of an embedded system with its own SoC and interconnects those in a network-on-chip (NoC). This requires radically new OS concepts for a parallel execution model. Each SoC runs tailor-made software components; traditional OS duties like memory protection, scheduling, or interrupt handling are no longer required for each SoC.

The NoC OS consists of hardware and software that administers application-specific components and handles SoC communication and synchronization. Based on our previous experience with feature modeling and AOP, we describe our approach to create an FPGA-based NoC OS and discuss its impact on embedded software development in comparison to traditional methods.

1 Introduction

Embedded systems today have to execute multiple tasks in parallel, even on small-scale, resource-limited processors. This poses a challenge for the system software for these devices, since multitasking and synchronization has to be performed on simple microcontrollers without proper memory protection, often while fulfilling hard real-time requirements.

In the area of automotive systems, OSEK\textsuperscript{1} [9] is a standard for a simple real-time OS environment implemented in commercial and open source products from different vendors. OSEK is a rather simple environment providing multitasking using a fixed set of tasks and priorities, interrupt handling and process synchronization, but no advanced system software capabilities.

In future systems, several applications have to be integrated onto a single microcontroller for cost reasons. Since OSEK-based systems do not provide mechanisms for the isolation of multiple processes running on top of it, AUTOSAR [1] was introduced to extend OSEK’s capabilities. This new standard adds support for memory protection, timing protection, and service protection to OSEK-based systems.

Considering the protection mechanisms available in AUTOSAR, the sharing of the CPU (timing protection), main memory (memory protection), exclusively usable resources (service protection by avoiding deadlocks) as well as fault containment (service protection by ensuring a fault in one application does not stop the entire system’s operation) are the resource sharing and protection mechanisms to be introduced when consolidating multiple OSEK-based application onto one embedded system.

However, the AUTOSAR approach does not require protection to be implemented. In fact, few of the systems currently in use employ memory protection mechanisms. Thus, the isolation requirements for a safe parallel execution of multiple tasks are still not necessarily satisfied in an AUTOSAR-based environment. In order to achieve reliable isolation for legacy software – independent of a specific software implementation – reliable mechanisms that enforce isolation are required.

The structure and management of these hardware mechanisms is the focus of this paper. We introduce ARCHES\textsuperscript{2}, an approach to building networked multiprocessor, application-specific, tailor-made embedded

\textsuperscript{1}German: “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen” – English: “Open Systems and their Interfaces for the Electronics in Motor Vehicles”

\textsuperscript{2}ARchitecture for Configurable Hardware structures in Embedded Systems
hardware/software systems.

With the availability of inexpensive, high-capacity and high-speed reconfigurable hardware today, hardware components are no longer scarce and expensive. When building a system based on an FPGA or ASIC, one can afford to put multiple processor cores, memories and peripherals onto one chip, building a networked multiprocessor system-on-chip (MPSoC) or network-on-chip (NoC) as a basis for real parallel execution of (legacy and newly developed) embedded software components. In an ARCHES-based system, each task in an embedded system is assigned its own, separate set of hardware resources. This removes the burden of implementing isolation from the operating system. In turn, rethinking and restructuring of the overall responsibilities and structure of an operating system for this distributed embedded architecture is required.

Using separate hardware for each task, the system also becomes much more adaptable. Since only one task runs on a given set of hardware, the requirements the software poses to the underlying hardware components – e.g., the minimum rate of instruction execution to meet timing requirements, the required amount of memory, and the set of peripheral units used – can be precisely determined. This enables developers to create hardware systems tailor-made for a specific task’s requirements. Since hardware today is developed using domain-specific hardware description languages (HDLs) like VHDL or Verilog, configuration in an ARCHES-based system can be achieved using methods adapted from our previous work on tailor-made infrastructure software [8].

Combining the approaches of separating and tailoring hardware resources, a radically new hardware and software structure is created that enables developers to concentrate on the task at hand instead of implementing operating system functionality.

The rest of this paper is structured as follows. Section 2 gives a short overview of OSEK-based operating systems. Section 3 discusses our approach to build multiprocessor SoCs. In section 4, structures for operating systems running on each SoC CPU as well as the required operating system for the overall NoC are discussed, followed by an outlook to using variability management for SoC hardware components. Finally, section 6 discusses related work and the paper closes with a conclusion and outlook to future work in section 7.

2 The OSEK OS Approach

OSEK [9] is a standard specification for a small real-time embedded operating system kernel. The structure of a typical OSEK system is shown in fig. 1. The OSEK kernel itself only provides mechanisms for multitasking, synchronization and interrupt handling. Device drivers are not a part of OSEK, but have to be implemented separately as part of the applications.

OSEK-based systems are configured statically, i.e., the number of tasks, resources, etc., is defined at compile time and does not change during the run time of a system. All modules of an OSEK system are typically linked into one large executable for the target system.

The system is written in C and uses several macro definitions to define system-level concepts like tasks and interrupt service routines:

\begin{verbatim}
DeclareTask(myTaskName);
TASK (myTaskName) {
    do {
        ... /* Task code */
    } while (!abortCondition);
    TerminateTask(); /* or ChainTask(...) */
    ...
}
ISR (myIsrName) {
    ...
}
\end{verbatim}

2.1 Multitasking

OSEK defines two task models, basic tasks and extended tasks. A basic task never waits for external events and can only assume a task state of ready, running, or suspended. An extended task, in addition, may wait for events and can enter an additional state, waiting. The task state diagram for extended tasks is shown in fig. 2.

A task can change its own state using the Terminate and Wait functions and it can influence the state of other tasks using Activate and Release.
2.2 Synchronization

Synchronization objects in OSEK are called resources. A resource is simply a binary semaphore that controls access to a given resource. Critical sections in OSEK are secured using the GetResource and ReleaseResource functions:

```c
TASK(controlTask) {
    GetResource(myResource);
    /* critical section */
    ReleaseResource(myResource);
    TerminateTask();
}
```

In addition, OSEK implements a priority ceiling protocol to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections.

2.3 System Parameters

Definitions of the parameters of the overall system are contained in OIL (OSEK Implementation Language) [10] definition files. These files specify the details of system generation. An example OIL file looks like this:

```c
OIL_VERSION = "2.5";

CPU myApplication {
    OS {
        STATUS = EXTENDED;
        ...
    } : "OS description";

    TASK taskOne {
        PRIORITY = 10;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = TRUE;
    };

    TASK taskTwo {
        PRIORITY = 11;
        SCHEDULE = FULL;
    }
}
```

Here, a system using extended task states is defined that uses two tasks. taskOne is automatically run at system startup with a priority of 10, while taskTwo is activated by an ALARM when the defined counter overflows. In addition, a resource myResource is defined.

2.4 Communication

Communication between tasks in an OSEK system is specified in an extension called OSEK/COM. It consists of an interaction layer which provides communication services for the transfer of application messages, a Network layer which provides services for the unacknowledged and segmented transfer of application messages and a Data link layer interface which provides services for the unacknowledged transfer of individual data packets over a network. Since OSEK/COM already provides primitives for IPC, the application software is abstracted from the network infrastructure by a lightweight layer of communication software.

3 Multiprocessor SoCs

Current chip manufacturing technologies allow large-scale integration of components. Today, it is possible to integrate not only the components of a complete embedded computer system on a chip as a system-on-chip, but rather to put many of these SoCs onto a single IC to create networks of SoCs. This advance in technology makes it possible to assign each task its own Soc in the ARCHES system.

3.1 Systems on Chip (SoCs)

SoCs integrate the functionality of a complete embedded computer system – CPU, peripherals, memory, etc. – on a single chip. Well-known examples are the ubiquitous microcontrollers at the heart of many embedded systems. When using programmable hardware components like FPGAs, an embedded systems designer can create a SoC tailored to a given specification. This hardware is today usually being described as soft cores using
a hardware description language (HDL) like VHDL or Verilog.

Figure 3: A simple System-on-Chip

Figure 3 shows a typical SoC. This system consists of a CPU core, three peripheral units (UART, pulse-width-modulation output and SPI bus) and the system memory containing executable code and data. All components are connected using a central bus system that transfers addresses, data and control information.

3.2 Networks on Chip (NoCs)

In all areas of computing, the use of multiprocessor and multicore systems is increasing as a consequence of limitations in achievable raw processor speed. This poses a difficult problem for general-purpose operating systems, since the load behavior of such a system is generally unpredictable and many software products today still only make use of a single thread of control.

In many embedded systems, like those based on the OSEK specification, however, the overall system functionality is predetermined and does not change during system runtime. It can thus be decomposed into several concisely defined tasks. As a consequence, many embedded systems can be adapted to use a multiprocessor system.

Due to ever increasing integration densities, current FPGAs and ASICs allow for the integration of more than one SoC on a single chip. In [3], the sizes and relative amount of chip area required for some example CPU and peripheral cores and the achievable clock frequencies for two common FPGA types from Xilinx, Inc, are given. On larger capacity FPGAs like the Virtex IIpro, each CPU and peripheral component uses less than 10% of the available chip space, so integrating multiple SoCs onto a single chip is feasible.

To enable communication between the various SoCs on one chip, the system has to provide a communication infrastructure.

Figure 4: A Network-on-Chip

A typical NoC is shown in fig. 4. Here, three SoCs are integrated onto one chip, communicating via a dedicated network. The structure of the network depends on the latencies and throughput required for the overall systems; implementations include bus- and ring-based structures as well as crossbar switches for systems requiring high throughput.

4 Restructuring the OS

The ARCHES approach combines the simplicity of the single-task-per-processor model with the flexibility of a distributed and networked environment by making use of multiple processor and peripheral soft-cores for FPGAs and assigning single, separate tasks of the overall system to separate SoCs. This allows the embedded software developer to use the single-task model for each processing unit and thus simplifies software development for embedded NoCs while providing a system that is able to execute multiple tasks in parallel.

By assigning each task in an embedded system its own SoC, several traditional operating system duties are no longer required in the infrastructure software for each SoC. In the following paragraphs, OS functionality that is dispensable in each SoC is discussed based on the functionality provided by an OSEK system.

4.1 Dispensable Functionality

Scheduling

Since each task is assigned its own processing unit, sharing of the CPU resource is no longer required – in fact, the software on each SoC has complete control over all of the SoC’s hardware.
This enables us to remove preemptive scheduling from each SoC’s infrastructure software. In turn, this will make it easier to estimate the processing power required for the related task. Thus, each SoC can be clocked down to the required minimum frequency to conserve energy.

In addition, real-time properties of a system are easier to determine and can be verified using tools like worst-case execution time (WCET) analysers.

When using the ChainTask functionality in OSEK, a system implements cooperative multitasking. A task calling the ChainTask system function puts itself into the Suspended state and activates another task. In an ARCHES system, this either implies putting the current SoC to sleep and sending a message to the SoC that owns the task to be activated or the replication of OSEK’s cooperative multitasking capabilities.

The SoC that is being suspended can then use power management technologies to conserve energy until an event (interrupt) occurs.

**Interrupt Handling**

In a system executing only a single task, scheduling using a timer interrupt is no longer required. In addition, an application may choose to implement peripheral handling using busy waiting. While this behavior is usually strictly objectionable in multitasking environments, it is useful in an ARCHES environment, e.g. in order to reduce latencies in device handling.

Interrupts can still be used to implement functionality local to a SoC. For example, an application might require a timer or asynchronous notifications from several peripheral units. Interrupts do not, however, preempt the running task from the CPU except for running an interrupt handler.

Since interrupt handlers are defined using the ISR macro in OSEK, compiler tools can be used to extract interrupt handler functions in legacy software and adapt the interrupt dispatch on each SoC accordingly.

**Isolation**

By constraining each SoC to run a single task, memory protection is no longer required on each SoC. This simplifies the software as well as the hardware for each SoC, since no MMU and virtual memory management software is required.

This physical isolation between tasks is, however, only possible if each SoC is connected to a separate memory instance. This is possible for small-scale embedded systems, since small amounts of memory can be implemented on an FPGA as block RAMs. If a common separate RAM is to be used for all SoCs, resource sharing and isolation mechanisms have to be provided in a central location.

**Synchronization**

Synchronization in the SoC itself is only required if interrupt service routines are used that operate on data structures common with the sequential task running on that SoC.

Otherwise, synchronization is only required when accessing common resources of the NoC. Requesting and releasing a resource can then be handled using NoC IPC mechanisms.

**Device Drivers**

Device drivers are still required for SoCs that communicate with peripherals (however, the special case of compute-only SoCs is also imaginable). Device drivers are, however, tightly integrated with the specific task of the SoC and can thus be implemented as library functions. This removes the requirement of building an API structure for driver management and, in turn, can help to reduce the executable code size.

The overall functionality required for the remaining system software for each SoC resembles earlier developments in application-specific operating systems like exokernels and related library operating systems [4]. By building a tailor-made version of libraries for each SoC, the code size can be efficiently reduced.

**4.2 Components of a NoC OS**

The remaining tasks for an operating system governing the overall network-on-chip consist of providing means for the various SoCs to communicate, synchronize access to common resources and to administer common resources for the NoC. In most distributed systems, synchronization functionality is implemented in each single processing component since communication in a wide-area distributed system is difficult to control centrally. In a controlled environment that provides high bandwidths and low latencies, like the networks that can be built on an FPGA or ASIC, one can afford to move large parts of the OS functionality into specialized central hardware components, thus forming an overlay operating system structure for the network-on-chip.

The following paragraphs describe the functionality required in a NoC OS, shown in fig. 5, that coordinates the complete embedded MPSOC system. The ARCHES architecture does not specify if this functionality has to be implemented in hardware or in software, so that components of the NoC OS can be implemented as distinct
IPC

The basic mechanism for inter-task communication in the ARCHES system is implemented using message-based IPC. The IPC send and receive mechanisms are asynchronous, thus, a task on a SoC can continue working without having to wait for a reply. The NoC OS IPC component then takes care of buffering messages.

Based on the use of the OSEK/COM API by the applications, communication structures of the overall system can be analyzed. If IPC between specific SoCs in a system requires a high bandwidth or low latencies, the ARCHES system is configurable to include a separate communication path to connect these SoCs, like shown between SoC 3 and 4 in fig. 5. Thus, the on-chip communication infrastructure can be tailored to the requirements of the applications.

Synchronization

Synchronization between multiple tasks is layered on top of IPC messages. All synchronization requests received by the system’s IPC component are handed over to a dedicated synchronization unit that grants or denies access to a specified resource and sends this decision back to the originating SoC using IPC.

Energy Management

Since in an ARCHES-based system, more CPU cores are active at the same time compared to a single CPU system, energy use of the system is a major concern. While application-specific tailoring of SoC components and speed is an important step to reduce power usage, more advanced methods are required.

4.3 Mapping OSEK Concepts to ARCHES

Since OSEK provides very few abstractions, implementing an application using OSEK APIs will be straightforward in an ARCHES system. The basic concept of tasks can be directly mapped to a task running on one SoC; as a consequence, each SoC is assigned a system-wide unique number corresponding to the task ID. IPC, then, can use this ID to address tasks running on different SoCs, the interconnect network of the NoC then uses the ID for routing decisions. The overall communication structure can be deduced from the use of OSEK/COM communication facilities.

Handling interrupts is also straightforward. Since OSEK interrupt service routines are marked with the ISR keyword and the compiler toolchain takes care of setting the interrupt vectors, the only modification that has to take place here is the assignment of an ISR to the correct task and the setting of interrupt vectors according to the hardware of the specific SoC.

This simple mapping will make it possible to automatically map legacy OSEK-based source code onto an ARCHES system. Creating the compiler and distribution tools required is thus an important step in the future evolution of the ARCHES system.

5 Variability

In order to build configurable embedded systems, an overview of the possible configuration options for hardware and software is required. Configurability in system software is a well-known topic. Of interest to ARCHES are the possibilities to configure hardware and to create a hardware configuration based on the requirements of the software running on top of it.

In ARCHES, we have to distinguish between configurability on two different levels. On the one hand, the options available for each of the SoCs is of interest, on the other hand, configurability of the overall NoC to application requirements is also important. In the following paragraphs, we discuss the various configuration options available in ARCHES.

5.1 Configurable SoCs

Systems-on-Chip consist of a processor, memory and several peripheral units. Data exchange between CPU and peripherals occurs using memory-mapped registers...
and (optionally) hardware interrupts. The structure of a typical AVR microcontroller is shown in fig. 6.

Figure 6: Atmel AVR Microcontroller Structure

Besides the CPU core (in the grey box), it offers two configurable serial interfaces, programmable digital port I/O pins, a JTAG debug interface, in-system programmability, and a watchdog timer. However, any given application will only use a subset of the large options available. This leads to a large amount of different chip configurations offered by the microcontroller manufacturer; the task of an embedded systems designer who relies on discrete components is then to select the chip that is best suitable from the many available options.

In ARCHES, a SoC will be configurable to the requirements of an application and its OS components. Components of the SoC are configurable in various dimensions: the number of instances of a functional unit (e.g., two serial interfaces), the size of a specific unit (1024 Bytes of RAM) and the speed of a unit (8 MHz CPU clock, maximum RS232 speed of 115.2 kbps).

While the number and size of units has direct influence on the size of the SoC, the speed of a unit determines a large part of the energy consumption of the SoC. Determining the exact relationship between the number and size of components as well as the relation between speed and energy use is a necessary evaluation for the practical usability of the ARCHES approach.

5.2 Configurability in the NoC OS

ARCHES-based systems are configurable in many ways. In addition to tailor-made configuration of the system software and the SoC processor and peripheral components, the architecture of the NoC OS itself is configurable to the overall application’s requirements. Here, communication bandwidths and latencies are important parameters that determine the performance of the system.

In turn, the communication infrastructure is adaptable to the requirements of the system. While a simple, low-cost implementation may choose to implement a bus- or ring-like structure to interconnect all SoCs, systems with high bandwidth or low latency demands may choose to implement a \( n \times n \) interconnect structure using a crossbar switch. In addition, a mixture of the several approaches is possible by implementing direct communication links between specific SoCs while running the general communication of a system through a centralized structure.

6 Related Work

6.1 Hardware Operating Systems

The idea of implementing an operating system using hardware components exists for quite some time. In [14], BORPH, a system for high-performance reconfigurable computing is described. It explores the design and implementation trade-offs of an operating system for FPGA-based reconfigurable computers. However, BORPH-based systems are designed for high-performance computing applications. While some of the concepts could be applied to embedded systems, the overall architecture is too complex, especially for small-scale systems.

An operating system structure for reconfigurable computing is the focus of [16]. Here, the authors analyze components of a classical OS like scheduler, memory management, I/O handling, and IPC as to their suitability for an implementation in hardware. The authors were able to implement some of the components in hardware and built a first prototype system, but no quantitative evaluation of the results was performed. This approach implements all functionality of an embedded system in hardware, whereas ARCHES uses hardware structures to control a distributed hardware/software system.

Interprocess communication in hardware is the topic of [5]. The overhead of IPC operations is analyzed and hardware structures to accelerate IPC operations are developed to support a real-time kernel environment. A similar implementation will be a good starting point for a hardware-based implementation of IPC for the NoC OS.
6.2 HW/SW Co-Configuration

The TOPPERS project at Nagoya University, Japan, creates an open platform for embedded and real-time systems. Part of the project is hardware/software co-configuration of a multiprocessor SoC for the TOPPERS RTOs [15].

The system runs a small RTOS with statically assigned applications on each processor of a system; each system resource is owned by a specific processor, but is made accessible to other processors. The architecture of the hardware is co-configured according to the requirements of the RTOS.

While this approach is rather similar in structure to our ARCHES system, it does not go as far as eliminating most OS functionality from each processor and assign each task a separate CPU. Unfortunately, the only information published in English is a slide set from a 2004 conference, so it is hard to determine if any real work was done in the area of co-configuration.

6.3 Multiprocessor Systems-on-Chip

IBM Cell

The Cell processor [6] is a heterogeneous multiprocessor on a chip. It consists of a 64-bit PowerPC core and eight so-called synergistic processing elements (SPEs), 128 bit RISC CPUs with 256 kB embedded SRAM, connected by the Element Interconnect Bus (EIB), a circular ring infrastructure comprised of four 16 Bit wide unidirectional channels.

Cell uses on-chip resource management implemented in hardware. The Resource Allocation Manager manages access of the SPEs to system memory and I/O interfaces by handing each SPEs access tokens relative to their soft realtime requirements.

While the Cell architecture is an interesting example of a MPSoC, its use is more interesting in high-performance computing and media processing applications due to its high cost and energy consumption. However, Cell has still found widespread use in the high-end embedded area as processor for the Sony Playstation 3 games console.

In addition, current Cell implementations lack the flexibility to adapt the SPE cores to a specific application – however, the Cell specification is flexible and cores could be adapted to specific uses.

Parallax “Propeller” CPU

The Parallax Propeller CPU [12] is an eight-core, 32-bit system intended for use in small-scale embedded applications that require parallel execution of tasks.

Each core has 512 32-bit words of local RAM, a high-precision timer and an interface to the common communication infrastructure. Resources common to all cores are I/O pins and 8 kWords of RAM.

All cores communicate in a round-robin fashion using a “hub” that connects each core for two clock cycles in turn to the central resources. Synchronization in the Propeller is performed using special assembly language constructs.

While the Propeller is an interesting concept for embedded software development, it is rather inflexible compared to our approach. The RAM and CPU power in each core is very limited and not adaptable and the communication bandwidth is limited by the round-robin communication approach of the central hub. Outside of the hub, which synchronizes access to common resources, support for operating system primitives in hardware is limited. Only some simple sort of synchronization is available through a specialized machine instruction.

Xmos XS1-G

Xmos’ XS1-G [17] is a multi-core, multi-threaded processor designed for embedded control applications. Using so-called “software defined silicon”, software is used to implement functionality that formerly required a hardware peripheral core. The XCore tile is the basic structure of the XS1-G, it consists of a 32-bit, 8-thread-capable RISC CPU with 64 kBytes of RAM, 8 kBytes of OTP ROM and 64 programmable I/O pins. XCore tiles can be interconnected on-chip using XLink channels. Inter-thread and inter-core communication is handled internal to the processor using specific machine instructions to provide intra-CPU thread-to-thread communication, on-chip communication via XLink switches and even off-chip communication using external ports.

While the architecture of the XS1-G is impressive, it lacks the flexibility and adaptability of our ARCHES approach, since the number of threads and the amount of memory available is fixed in silicon. However, the idea of software-defined silicon is interesting and can also make a useful extension of our concept.

7 Conclusions and Future Work

In this paper, we presented ARCHES, our approach to create configurable embedded hard- and software

3The Xmos architecture is based on the ideas of the innos Transputer CPU. In fact, one of Xmos’ founders is David May, the architect of the Transputer and the OCCAM language.
systems. Compared to current software-level integration technologies based on OSEK like AUTOSAR, the ARCHES approach has the advantage of providing real physical CPU and memory isolation while creating an environment for true parallel execution of tasks.

This allows the embedded system designer to concentrate on the task that is to be solved instead of creating system software. Each task can initially be implemented independent of all other activities in the system regarding its CPU and memory allocation; synchronization is only required when accessing resources of or waiting for events from another task.

7.1 Automatic System Generation

OSEK provides information about the task and resource configuration of an embedded system in the OIL files and through the TASK and ISR macros used in the C code of a system. Using this information, combined with an analysis of the generated binary code, an estimate of the memory and interrupt resources required for a single task could be made in order to automatically generate a SoC for that task with tailored memory sizes.

In addition, the speed of that SoC can be determined by using a WCET analysis tool with specified real-time constraints. By limiting the speed of an SoC to the required minimum, the overall energy consumption of the system can be efficiently reduced.

Further possibilities for automatic tailoring a system would require more detailed code analysis. For example, an analysis of the defined interrupt service routines and memory mapped I/O locations used could give an indication about which set of peripherals the current task is using.

For automated generation of a combined hardware/software system, our previous experience with application analysis for tailoring infrastructure software [13] will prove useful.

7.2 Application-Specific Processors

Depending on the specific task, the structure of a CPU to execute an application’s binary can differ significantly in order to obtain optimal results. While all (Turing-complete) CPU implementations are able to execute code compiled from a high-level language – assuming a compiler is available and the CPU is able to address a sufficient amount of memory – there are applications that may benefit from special CPU hardware, like a floating-point unit or a hardware multiply-and-accumulate ALU, to operate efficiently. Thus, a single CPU type is usually not optimal for all applications in a ARCHES system.

However, the system does not require that the same CPU core is used in every SoC; one can easily imagine a heterogeneous NoC using CPUs selected according to a specific application’s requirements. Thus, application-specific processors can be easily integrated.

While our current system structure assumes the use of “off-the-shelf” CPU cores, for which compiler toolchains exist, more detailed analysis of each task can give hints to adapt the CPU of a SoC itself. Adaptable parameters include the width and number of CPU registers and buses, the instruction set, and the availability of functional units like FPUs. Integration of these application-specific processors [11] into an ARCHES system will show valuable improvements in size or speed of the various SoCs.

The current concept of using identical processor cores for each SoC, however, can be useful when the available chip area is restricted. Since all CPU cores share architectural features, the sharing of specific components – a special-function ALU or a floating-point unit like the one in Sun’s OpenSparc T1 multi-core CPU – could be used to reduce the size of the circuit.

7.3 Energy, Cost and Technology

Various projects rely on reconfigurable hardware (FPGAs) to build configurable embedded systems. While these systems provide the advantage of run-time reconﬁgurability, this is currently a highly experimental feature. Relying on run-time features of an FPGA means, however, that for large production runs of a chip, much cheaper ASIC technology cannot be used. With ARCHES’s static configuration approach, however, the hardware portion of an embedded systems is created at design time, making an ASIC implementation of a tailor-made embedded hardware/software system possible.

Our approach has to overcome some more challenges. Since every task in the system uses a dedicated CPU, busy waiting for events is an admissible policy in our system. Thus, the OS has to make provisions for enhanced power management for the complete distributed system. This can be achieved by frequency scaling and employing sleep modes for single SoCs that are triggered by software on the SoC. Waking up a sleeping core is a task of the NoC OS, which is enabled to monitor relevant state changes for each core.

7.4 Flexible HW/SW-Interfaces

Systems like Xmos XS1-G and Parallax Propeller show that it is feasible to reduce the dependance of hardware components in an embedded software. Using suf-
ciently fast CPUs, much functionality that formerly required a hardware core implementation is now implementable as software components that drive simple digital I/O pins.

Several examples show that it is possible today to create peripheral controllers that consist mostly of software with only the minimal support in hardware (e.g., simple I/O port pins). Using AVR controllers, peripherals like a low-speed USB device controller, a VGA video signal generator, and an SPI controller have been implemented as hand-tuned software for low-cost microcontroller cores.

This approach has several advantages. One obvious benefit is that software developers are now enabled to create hardware components. Since the number of C programmers currently exceeds the number of embedded hardware developers for VHDL or Verilog by a factor of 100 [7], it will be easier to find developers. Another advantage shows when implementing a system in an ASIC. Here, bugs in a hardware protocol implementation are impossible to fix in the field. Using a software implementation, however, only an upgrade of the firmware is required to fix bugs in an otherwise static silicon structure on a chip.

The ARCHES system can form an ideal basis for implementing such a software-dominated embedded system. Since every task and its associated resources is encapsulated in its own SoC, the overall system does not care of implementation details. Thus, a given task can either use hardware peripherals or peripherals implemented in software, trading chip area against execution speed.

The exact implications of software-implemented hardware components on chip size and energy usage as well as overall system stability will have to be investigated in detail; this is an interesting topic for future research.

References


