Prototyping Networked Embedded Systems

Josef Fleischmann, Technical University of Munich
Klaus Buchenrieder, Siemens Corporate Technology

Sophisticated consumer devices that support multimedia—personal digital assistants, network computers, and mobile communication devices—pose challenges for embedded-systems designers. The low-cost, consumer-oriented, fast time-to-market mentality that dominates embedded-system design today forces design teams to use hardware-software codesign to cope with growing design complexities. New codesign methodologies and tools must support a key characteristic of next-generation embedded systems: the capability to communicate over networks and adapt to different operating environments.

NETWORKED EMBEDDED SYSTEMS

Two emerging classes of embedded systems that operate across networks challenge designers:

- *Multifunction systems* execute multiple applications concurrently.
- *Multimode systems* offer users alternative modes of operation.

Editors: Jerzy W. Rozenblit, University of Arizona, ECE 320E, Tucson, AZ 85721; jr@ece.arizona.edu; and Sanjaya Kumar, Honeywell Technology Center, MS MN65-2200, 3660 Technology Dr., Minneapolis, MN 55418; skumar@htc.honeywell.com

With their tight time-to-market and cost constraints, networked embedded systems pose new challenges to designers.

Multifunction systems

Emerging embedded systems concurrently execute multiple applications, such as capturing video data, processing audio streams, and browsing the Web. These systems must often adapt to changing operating conditions. For example, network bandwidth and loss rate may vary, so multimedia applications must adapt, modifying video frame rate in response to network congestion. They thus implement a trade-off between quality of service and network bandwidth.

Likewise, audio applications must, as one of their functions, apply different compression techniques, depending on network load and quality-of-service feedback from client applications.

Multimode systems

Embedded systems designers must also deal with multimode systems: those with several alternative modes of operation, such as a mobile phone, which performs a single function but can change the way it operates to accommodate different communication protocols.

Flexible, multimode devices are also mandatory for applications like electronic banking and electronic commerce. Depending on the type of connection and the security level required, devices must apply different encryption algorithms when transmitting data.

Designers must also deal with other, less obvious multimode systems. The rapid evolution of Web-based applications, for example, causes requirements for devices like set-top boxes to change within months. For certain application domains, designers can alleviate the problem of short product lifetime by designing hardware and software system components that users can configure or upgrade after production. True, the average PC user is used to downloading software upgrades and bug fixes, but most embedded devices don't yet support such tasks.

Recently, remote administration of electronic products over the Internet has become an important feature: Printers or copiers with embedded Web servers are already available. Via reconfigurable hardware components, vendors can change hardware-implemented functionality after installing networked devices at the customer site.

Codesign challenge

In view of these innovative types of embedded systems, researchers have proposed several hardware-software codesign tools and frameworks, and industry design teams have adopted some. There is, however, a lack of methods and tools to investigate the issues raised when designing configurable hardware-software systems. Therefore, we are developing a complete design environment for embedded systems that includes dynamically reconfigurable hardware components. We based this environment on Java, using it for specification and initial profiling, as well as for the final implementation of the system's software components. We chose Java because it is a simple-to-use, object-oriented language.

SYSTEM-LEVEL DESIGN

Several trends influence the nature of embedded systems development and shape requirements for the optimal development tool.

Design trends

First, designing an embedded system's digital hardware has become increasingly similar to software design. The widespread use of hardware description languages and synthesis tools makes circuit design more abstract. Market pressures to reduce development time and effort encourage abstract specification as well, and promote the reuse of hardware and software components. Therefore, a specification language should provide a comfortable means for integrating reuse libraries.

Second, object-oriented programming has proven to be an efficient paradigm for the design of complex software systems. Although OO may imply performance loss, it comes with significant benefits: Not only does OO provide a better means for managing complexity and reusing existing modules, it also reduces problems and costs associated with code maintenance. These benefits far outweigh the performance loss.

Third, in embedded system design one major trend is to increasingly implement functionality in software. Doing so achieves faster implementation, more flexibility, and easier upgradability and customization with additional features. Unlike hardware, software incurs no manufacturing costs per se, although the costs of software maintenance cause increasing concern.

The Java solution

For these reasons, we chose Java as the specification language in our design flow. It is a clean, object-oriented, versatile language of moderate complexity. Because Java has built-in primitives for handling multiple threads, it well supports the concurrency and management of different control flows. Based on our experiences with different projects, we found that new applications can be rapidly developed in Java.

The design of networked embedded systems requires support for a diverse feature set that includes Internet mobility, network programming, security, code reuse, multithreading, and synchronization. However, Java has not been designed for specifying systems with hard real-time constraints, so we are investigating extensions and restrictions to the language that will help it do so.

The need for communicating with embedded systems over the Internet pushes more designers toward Java.

Meanwhile, the need for communicating with embedded systems over the Internet pushes more designers toward Java, which has recently been adopted as the premier design platform for implementing set-top box applications. Set-top boxes are a rapidly growing embedded-systems market that promises to reach millions of homes.

DESIGN EXPLORATION

We developed a cosynthesis method and prototyping platform specifically for embedded devices that combine tightly integrated hardware and software components.

Assigning tasks

We use cosynthesis to make the most efficient assignment of tasks to either software or hardware. Our cosynthesis method begins with an initial Java specification of the desired functionality. Software profiling by the Java virtual machine then identifies bottlenecks and computation-intensive tasks. Using a graphical visualization tool that displays each task's relative and absolute execution times, the designer quickly uncovers an application's most computationally demanding tasks.

Next, the designer seeks candidate tasks for hardware implementation. They make these selections based on profiling results and a reuse library of available hardware components. The designer uses a high-level synthesis tool to transform Java methods into register-transfer-level VHDL (Very High-Speed Integrated-Circuit Hardware Description Language). At the same time, the tool generates an appropriate interface description for each hardware block.

The target architecture for synthesis is the prototyping and exploration platform shown in Figure 1. It handles software and hardware parts of the design separately. A runtime environment implemented in software on a PC helps prototype software, while an additional configurable hardware extension—the dynamically reconfigurable, field-programmable gate array (DPGA) board—handles hardware.

In the runtime environment, the Java virtual machine forms the core component of the software execution engine. A database stores information about the classes and methods used by the design under test. Software synthesis takes the form of generated bytecode: compiler-encoded, platform-independent code. The virtual machine contains the necessary interface mechanisms (the hardware wrapper and device driver) to communicate with hardware modules.

This approach yields a smooth migration from a pure software implementation to a mixed hardware-software system without modifying the Java source code. Further, adding components for hardware object handling to the Java virtual machine, and interfacing it to an external hardware component, automates much of the configurable hardware device's management.

Prototyping the design

For the design exploration platform, we use a prototyping board connected to a PC via a PCI bus. The board consists of a DPGA chip and local memory. The DPGA offers short reconfiguration times and full access to an implemented cir-

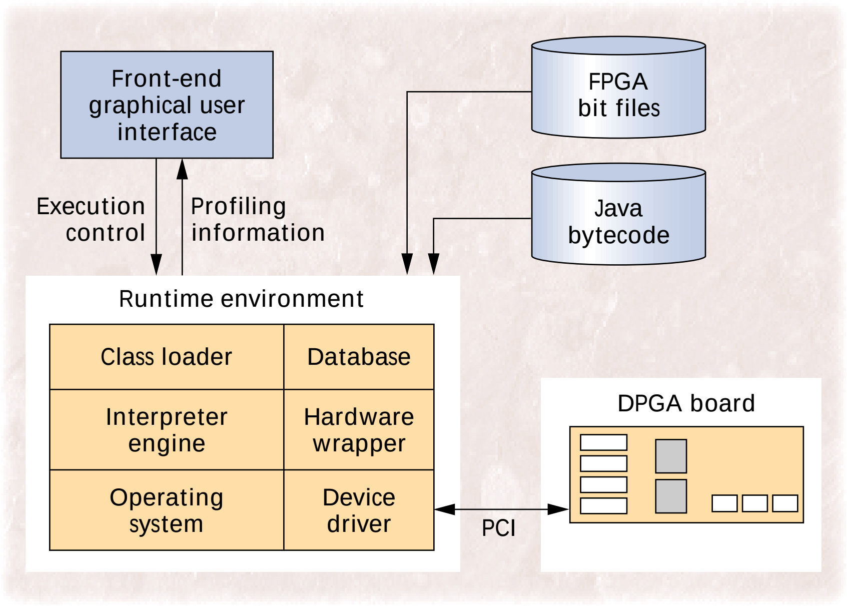

Figure 1. Design exploration platform for networked embedded systems. The runtime environment reads the FPGA bit files and Java bytecode inputs, activating the hardware-call module whenever the control flow reaches a hardware method. At this point, the data is sent to the DPGA board, which emulates the appropriate hardware device.

cuit's internal registers, which allows the mapping of multiple hardware objects onto a single chip. Further, parts of the chip can be reconfigured even when other parts are operating, allowing execution of multiple methods on the DPGA board when running the system prototype.

In practice, the runtime environment in Figure 1 reads in a table with the desired function partitions routed to either hardware or software implementations. The environment directly handles those tasks executed as software on the Java virtual machine. For hardware modules, it configures the DPGA (with information from the DPGA bit files) and manages its communication with the DPGA.

During application execution, the interpreter must activate the hardware-call module whenever the control flow reaches a hardware method. Depending on the DPGA board's current state, a hardware call can trigger one or more actions:

- complete or partial reconfiguration of the DPGA,
- the transfer of input data to the board,
- the transmission of an enable signal that in turn starts the emulation of the hardware design, and
- the transfer of data back to the calling thread.

These procedures are implemented in the hardware wrapper shown in Figure 1. Further, we implemented a strict synchronization mechanism that currently allows only one thread at a time to access the DPGA board.

Running sample applications on this prototyping platform lets us validate the mixed hardware-software design against the initial, software-only specification. We found that the initial software specification cannot provide enough details for optimizing the hardware-software boundary because there is too little detail about the final implementation.

During prototyping, we are especially interested in different protocols and interface mechanisms. The working prototype provides more reliable information about the final system's performance. Therefore our approach combines profiling with more detailed information from the hardware-software synthesis and from the prototype execution to optimize the design.

OPTIMIZATION

Like all embedded-system designers, we seek to maximize performance within the constraints of limited hardware resources. So during optimization, we focus on those parts of the design that could be alternatively implemented in software or hardware, and their corresponding interfaces. Our prototyping environment gathers characteristic information about hardware and software modules and stores it in a library of reusable modules. The most important parameters measured are

- execution times of a module for both hardware and software implementations,
- required area for hardware implementation, and
- the specific interface costs (in terms of additional execution time for data transfer, hardware area, or software code).

In systems that depend on both hardware and software, efficient interface design is crucial to achieving maximum performance. For this reason, our optimization process emphasizes the efficient design of interfaces while searching for an optimum mapping of modules to hardware and software.

Mapping to hardware and software

To model interfaces, we use the basic structure and interface types shown in Figure 2. The interface model contains a communication channel for software modules, an interface for data exchange between the hardware and software domain, and a two-layer communication channel for hardware entities. Layer 1 represents a direct communication link that is implemented on-chip. Layer 2 represents communication via SRAM.

During design optimization, the prototyping environment calculates the interface overhead for the current mapping of modules to the hardware and software partitions. If we map tightly coupled modules to hardware, communication is implemented through on-chip registers or through shared memory, using static RAM.

Our prototyping environment optimizes a system design for minimum execution times within the constraints of limited DPGA chip area and communication bandwidth. Accordingly, to represent the quality of any implementation by a single numerical value, we use a cost function. Our cost function is the weighted sum of the squares of each cost—hardware, software, and communications.

For small designs, we assign partitions through an exhaustive-search algorithm, which finds the implementation that provides the cost function's minimum value. Due to exponential complexity, however, this approach is infeasible (runtimes are too long) for systems with more than 25 modules. Because partitioning belongs to the group of NP-complete problems, you can infer that a solution to one instance of the problem provides the solution to all of them. Thus we decided to choose a more suitable, heuristic optimization method: simulated annealing.

Simulated annealing

The simulated-annealing method models the physical process of melting a material and then cooling it so that it crystallizes in a state of minimal energy. This method has, however, been successfully applied to several problems in very large systems integration. Further, it is easy to implement for differing cost functions and usually delivers good results. The algorithm, a probabilistic search method that can climb out of a local minimum, consists of two nested loops:

- The outer loop decreases the current temperature according to a certain user-specified cooling schedule.
- The inner loop generates and evaluates several new partitions by shifting modules from one partition to another (depending on the current temperature).

The algorithm accepts moving a module from one partition to the other if doing so decreases the overall cost function. With a certain probability that depends on the current temperature, it may also accept cost *increases*. In this way, the algorithm can escape from a local minimum. The inner loop repeats until the algorithm detects a steady state for the current temperature. Depending on the temperature schedule and the stopping criterion, a trade-off between computation time and result quality is possible.

As we refined the method, we cut computation times significantly by choosing an intelligent initial partition—instead of a random one—before starting optimization. The algorithm then begins with a lower starting temperature, which reduces the number of iterations.

For example, the optimization process for a reasonably sized system with about 100 modules takes less than one minute on a Pentium II. Because simulated annealing is a probabilistic method, two runs of the algorithm may produce slightly different results. But with such short runtimes we can repeat the optimization process several times, thereby increasing the likelihood of obtaining a near-optimal result.

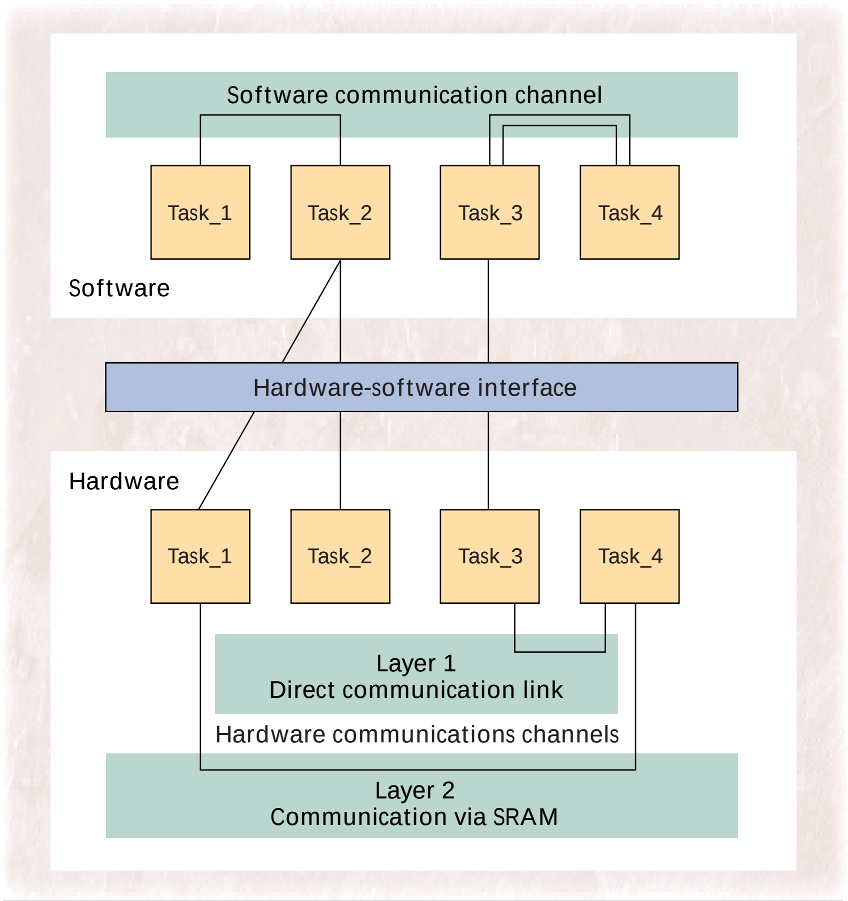
Software communication channel
Task_1
Task_2
Task_3
Task_4
Software
Hardware-software interface
Hardware
Task_1
Task_2
Task_3
Task_4
Layer 1
Direct communication link
Hardware communications channels
Layer 2
Communication via SRAM

Figure 2. Target system communication model. The software communication channel connects with the two-layer hardware communication channel through an interface that allows data exchange between the domains.

Java-based rapid prototyping of embedded systems offers several benefits. It allows system-level testing of novel designs and architectures using a flexible platform for development of both software and hardware components. Java-based prototyping also combines profiling results from the specification level and characteristic measures of the prototype implementation. Combining these data gives a more solid prediction of the final system's performance and cost, which makes possible a reliable optimization strategy. ❖

Josef Fleischmann is a research assistant at the Technical University of Munich. Contact him at Josef.Fleischmann@ei.tum.de.

Klaus Buchenrieder heads research in hardware-software codesign at the Central Technology Laboratories of Siemens AG in Munich. Contact him at Klaus.Buchenrieder@mchp.siemens.de.