ROLF ERNST,
TECHNICAL UNIVERSITY
OF BRAUNSCHWEIG

Putting it all TOGETHER

Component integration is one of the tough challenges in embedded system design. Designers search for conservative design styles and reliable techniques for interfacing and verification.

With the growing complexity of embedded systems, more and more parts of a system are reused or supplied, often from external sources. These parts range from single hardware components or software processes to hardware-software (HW-SW) subsystems. They must cooperate and share resources with newly developed parts such that all of the design constraints are met. This, simply speaking, is the integration task, which ideally should be a plug-and-play procedure. This does not happen in practice, however, not only because of incompatible interfaces and communication standards but also because of specialization.

Take, for example, a signal processing program that has been adapted to a specific digital signal processor (DSP) architecture by carefully rewriting the source code using special functions or subword parallelism, optimizing loops, data transport, and memory access. Reusing such a DSP program means either rewriting that code or reusing the whole DSP architecture or part of it, turning the original SW integration problem into a HW-SW integration problem. A crypto algorithm that runs on an application-specific instruction set processor (ASIP) is another example. DSP and ASIP architectures are great for reaching the performance and power consumption goals, but they make portability and, thus, reuse more difficult.

Unfortunately, compilers that could automatically adapt the code are not yet available—and designers are happy if they can avoid assembly coding. You may continue the list with HW accelerators, specialized memory architectures, buses, etc. Architectural variety and adaptation seem inevitable to reach demanding design goals for competitive systems. This is what's driving the revival of ASIPs. Thus, we will have to live with heterogeneous embedded system architectures and their corresponding integration problems. This holds for system-on-a-chip (SoC), as well as for larger distributed embedded systems.

On the other hand, there is a tendency toward SW integration. The automotive industry, for example, is accustomed to a business model where the supplier provides the electronic control unit together with the SW implementing a specific automotive function, such as engine control, dashboard, window motors, antilock breaks (ABS), and adaptive cruise control (ACC). Integration means that the zoo of about 50 control units is hooked up to automotive buses, which have to carry the entire communication load. This is an increasingly difficult task because of distributed automotive functions such as ACC. To reduce the number of control units, the software should be integrated on fewer control units, but this leads to the problem of certifiable SW process integration on shared processors using the OSEK automotive operating system (www.osek-vedx.org). This particular

Putting it all TOGETHER

problem was part of a Hot Topic session of the Embedded Software Forum at this year's Design Automation and Test in Europe (DATE) conference in Munich (www.date-conference.com). The automotive industry is just one example. Just think of SW integration in telecommunications, on home platforms, or mobile communication systems.

The main integration tools are the communication, and possibly memory, infrastructure, as well as the basic software, meaning the realtime operating system (RTOS) and communication software providing support for resource sharing and interfacing. To that list we can add application program interface (API) software, which increases portability.

INTEGRATION CHALLENGES

The three main types of design tasks in embedded system integration are:

- Component and subsystem interfacing
- System verification
- System optimization with design space exploration

The first two tasks are general design problems, whereas the latter depends on the cost and optimization pressure of an application. This article looks at the first two issues, which are also prerequisites to system optimization and design space exploration.

Interfacing is well developed at the RTOS level. There are SW-SW communication primitives such as queues (pipes) for message passing, shared variables, and semaphores for synchronization. These communication primitives separate computation from communication. The communication primitives are mapped to platform-dependent functions. This way, software can be ported more easily by eliminating the need to implement new communication primitives.

In contrast, the HW description languages in use today—VHDL and Verilog—support communication via

electric signals only. Porting HW components to a new design requires HW process adaptation. SW-HW communication uses drivers that, again, must be adapted to the HW protocol. Using similar communication primitives on both sides would make HW adaptation and driver development much easier.

Therefore, newer HW description languages—such as SpecC, a C language extension (www.SpecC.org), and SystemC, a C++ class library (www.SystemC.org)—extend HW communication to abstract primitives comparable to RTOS communication. With such primitives, the HW component function can be separated from its communication with other system components, similar to RTOS primitives. Integration can, therefore, focus on implementing the communication primitives, which might be reused for the integration of other components. This development of new languages is ongoing, but standards and first tools are available, with support from major EDA vendors.

Interfacing is necessary but not sufficient. The fact that the parts properly talk to each other does not mean they work together as required. This is a matter of semantics and target architecture performance. Both must be checked in system verification. Function verification focuses on the system semantics, which should be implementation independent. Performance verification should validate hardware parameters, processor resource sharing, and communication performance to detect performance pitfalls such as transient overloads or memory overflows.

Typically, both function and performance verification use prototyping or simulation (“virtual” prototyping). Prototyping uses a different target architecture, at least for parts of a design. For such parts, prototyping allows only function verification. Moreover, prototyping is expensive in terms of development time, and it has limitations as far as available parts or non-reachable environment conditions (just think of modeling a specific engine failure or car accident). This article, therefore, focuses primarily on simulation.

Although function verification of an embedded system may use untimed simulation, performance verification relates to timing and therefore requires timed simulation—i.e., simulation where events have a time label. Because timed simulation needs far more computation time, performance verification is a bottleneck. Therefore, abstract timing models for components that manage to reduce computation time receive much attention. Such models range from so-called “cycle-accurate” models, which model the system behavior clock cycle by clock cycle, to networks of abstract state machines, such as in

the Cadence Design Systems Virtual Component Co-design (VCC) simulator (www.cadence.com).

Does that mean all we have to do is develop faster simulation models and simulators? Not quite. Consider the example in Figure 1. A supplier has provided Subsystem 1, consisting of a sensor (Sens) that sends signals to a microcontroller (CPU) running a system of processes with preemptive scheduling (i.e., scheduling where the execution of one process can be interrupted for the execution of another process); one of the processes is P_1 . Subsystem 1 uses Bus A two times, to read the sensor data and to write the data to a HW component, which could be a peripheral device generating output signals periodically. The sensor signal is buffered in the CPU memory M_1 . The supplier provides the working subsystem and simulation patterns simulating the worst-case CPU load situation, including the worst-case execution time (WCET) of P_1 .

The integrator decides to share Bus A with DSP Subsystem 2, which consists of an intellectual property (IP) component that generates periodic output data (e.g., a filter or digital-to-analog converter) and a DSP running a fixed periodic schedule. A buffer is inserted at the DSP input to resynchronize the data stream. This integration task is typical. The integrator is now rightfully worried about the distortion that Subsystem 1 traffic injects to Subsystem 2, possibly leading to extended end-to-end system response times and buffer under- or overflow at the DSP input. The integrator has no idea of the internal subsystem function; only the worst-case simulation patterns are available.

Now, look at the bus load. Figure 1 demonstrates that the highest transient bus load leading to the worst distortion of the Subsystem 2 traffic is caused by the best-case execution time (BCET) of P_1 , which was not a corner case in subsystem design. It is therefore likely that this system

corner case will not be covered in simulation and the system might fail.

This example shows a fundamental performance simulation problem. Simulation patterns from function simulation are not sufficient, because they do not check for the non-functional dependencies of the two functionally unrelated subsystems. The subsystem corner cases are not sufficient as they do not match the system corner cases. The system integrator cannot generate new corner cases because he/she is not aware of what the corresponding worst-case subsystem behavior might be. To make the situation even more complicated, communication of Subsystem 1 in the example is distorted not only by the DSP subsystem but also by its own sensor-to-CPU traffic. Unfortunately, the typical bus standards introduce such non-functional dependencies.

Unlike standard software, such uncertain behavior is intolerable in embedded systems design especially when life-critical functions are involved. Firing an airbag 10 milliseconds late, for example, consumes roughly half the time it would take your head to hit the steering wheel.

Putting it all TOGETHER

Even if lives are not involved, however, such system failures can be costly. They can make products unmarketable, as people are typically not willing to accept an embedded system with the quality of PC software.

CONSERVATIVE DESIGN

One possible answer is to use integration techniques and strategies that avoid non-functional dependencies. The Time Division Multiple Access (TDMA) protocol assigns a fixed time slot to each logic communication channel—i.e., Sens-CPU, CPU-HW, IP-DSP—and remains unused even if the communication is not active. This way, each logic communication channel receives a fixed share of the overall bandwidth regardless of the other subsystems. The discrete time slots introduce jitter, but this jitter can be bounded and may already be considered in component design. This conservative technique is adopted both on the chip level, where it is used, for example, by Sonics Micronetworks, and in larger-scale systems such as the Time-Triggered Protocol (TTP) for safety-critical automotive and aerospace applications (www.ttgroup.org).

The TDMA technique can be applied to processor scheduling and extended all the way to software development, where the elegantly simple math describing TDMA performance can be used for a system-wide performance analysis and control, such as in the Giotto tool of UC Berkeley. (For more information about Giotto, refer to “Time-safety Checking for Embedded Programs,” by Th. Henzinger, Ch. Kirsch, R. Majumdar, and S. Matic, in the Proceedings of the Second International Workshop on Embedded Software (EMSOFT), *Lecture Notes in Computer Science 2491*, Springer-Verlag, 2002, pp. 76-92, or <http://www-cad.eecs.berkeley.edu/~fresco/giotto/>.)

However, conservative design with TDMA comes at a performance (and power) price. If short response times are required, or if the system reacts to non-periodic and burst events, or if the load varies depending on system scenarios, then the system must be significantly over-

designed. The problem is that even a small change in the conservative strategy dilutes the conservative properties. For example, in a round-robin strategy, which assigns unused slots to the next process or communication in line, you see the same non-functional dependencies, even though round-robin at least guarantees minimum performance that is equivalent to TDMA.

PERFORMANCE ANALYSIS

Before resorting to conservative design, you might take a closer look at more formal performance analysis. Statistical approaches do not seem adequate given the complex deterministic communication patterns. They do not capture very specific overload conditions and either can be risky or lead to overly conservative design.

Advanced embedded systems engineers are likely to be familiar with the formal methods developed for real-time computing, at least with rate-monotonic scheduling and analysis (RMS and RMA). RMA shows the principle of such formal methods. RMA abstracts from individual process activations (as used in simulation) to activation patterns. Based on these activation patterns and process WCETs, it derives schedulability and worst-case response times.

A host of work exists in the realtime computing community on schedulability and response time tests using activation patterns and WCET as input. The WCETs are typically simulated or measured. We’ve seen major progress recently in formal program analysis leading to the first commercial tools available for modeling program execution (www.absint.com). Some open issues remain, such as coupling effects in cache architectures and BCET analysis, but we may expect formal analysis solutions in the near future that could replace or complement measurement or simulation, provided enough investment is made in the EDA technology and processor models.

If such formal methods are available, why the need for conservative design? The main limitation is that these methods do not easily scale to larger heterogeneous embedded systems, such as that shown in Figure 1. They cover one processor or bus or, at most, a subsystem with homogeneous scheduling. There are proposals combining a few different scheduling strategies, such as RMS on a processor and TDMA on the bus (see, for example, “Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems,” by P. Pop, P. Eles, and Z. Peng, *Proceedings of the International Symposium on Hardware/Software Codesign (CODES02)*, pp.187-192, Estes Park, CO, 2002), but I know of no general coherent “holistic” approach covering a system like that in Figure 1.

Consider Figure 1 again. We could partition the system into locally scheduled communicating components grouped around Bus A, which has its own resource arbitration protocol. These components send and receive messages that can be combined into message streams.

Figure 2 highlights these message streams. With some relatively simple math, you can transform the message streams to activation patterns, such that the analysis results of the sending component is propagated to the analysis algorithm of the next component. This also works for buses. Such a transformation is called Event Model Interface (EMIF). Continue propagation and analysis until you have reached the output. Components can be analyzed if all input streams are available.

This way, global performance analysis becomes an event flow analysis problem. Loops in the flow, such as between the CPU and Bus A (bidirectional flow) are solved by iteration. Eventually, you can also calculate the required buffer size at the DSP input. (For more details, refer to "A Formal Approach to MpSoC Performance Verification," by K. Richter, M. Jersak, R. Ernst, *IEEE Computer*, April 2002, or www.spi-project.org). Other researchers have looked into flow-based global analysis with a somewhat different model ("Real-time Calculus for Scheduling Hard Real-Time Systems," by L. Thiele, S. Chakraborty, and M. Naedele, *Proceedings of International*

Symposium on Circuits and Systems (ISCAS 2000), pp. 101-104, Geneva, Switzerland, vol. 4, March 2000). Flow-based analysis has been applied to early practical examples from telecom, automotive, and multimedia, even though expert knowledge is still necessary and no easy-to-use tool is available yet.

WHAT'S NEXT?

Given the development of embedded system complexity, simulation-based performance verification seems to be slowly running out of steam. This worries people in safety-critical applications today and, with growing system complexity, this will be a

key problem for any integrator. Conservative techniques alone are no general solution for power consumption, cost, and performance reasons. Formal methods have many benefits as an alternative to simulation-based performance verification, but must be extended to global analysis methods adequate for heterogeneous embedded systems. The real medium- to long-term alternative appears to be conservative design versus analytical performance verification. Conservative techniques could be used where sound performance verification methods are not applicable or are for whatever reason inefficient (e.g., too wide bounds resulting from abstract formal models).

Timed simulation can still play a big role and appears inevitable when continuous time models are included to simulate the embedded system together with its physical environment. Given the advances in analytical methods, however, we should reconsider whether we put the most energy in improving timed event-driven simulation or instead invest more effort in formal methods for performance analysis of complex architectures. Q

ROLF ERNST's current research interests include embedded architectures, high-level synthesis, hardware/software co-design, and embedded systems engineering.

© 2003 ACM 1542-7730/03/0400 \$5.00

BIO