INTERNATIONAL JOURNAL OF ENGINEERING AND MANAGEMENT SCIENCES

www.scienceandnature.org

Short Communication

HARDWARE-SOFTWARE CO-DESIGN OF EMBEDDED SYSTEMS

Dwivedi Kartikey
Department of Electronics and Communication Engineering, Manipal Institute of Technology,
Udupi-576104, Karnataka, India

ABSTRACT
Most of today's gadgets and automobiles use embedded systems, which, in many cases, has taken over what mechanical and dedicated electronic systems used to do. Indeed, embedded systems appear in everything from telephone to medical diagnostics, climate controls, manufacturing to aviation and space sciences. Designing of embedded system capable to perform planned functions in time and cost is a hardware-software co-design issue as hardware and software influence each other. Our endeavour is to discover challenges and issues of co-designing hardware/software for the desired performance, cost and time objectives.

Keywords: Embedded System, Embedded System Design, Embedded Software

INTRODUCTION

Creating an embedded computer system which meets performance, cost and design time goals is a hardware-software co-design problem. In any embedded system, design of the hardware and software components influences each other. Embedded Software (ESW) design is just one, albeit critical, aspect of the more general problems of Embedded System Design (ESD or just ES). ESD is all about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight. This paper explores critical relationship issues between hardware and software architecture in the early stages of design and then surveys various analysis techniques used to define hardware/software requirements for hardware-software co-design. Later, we have analyzed design and synthesis techniques for co-design and related problems.

EMBEDDED SYSTEMS DESIGN

Embedded system design can be divided into four major tasks:

a. Partitioning the functions to be implemented into smaller, interacting pieces;
b. Allocating those partitions to microprocessors or other hardware units, where the function may be implemented directly in hardware or in software running on a microprocessor;
c. Time Scheduling the functions for their execution, which is important when several functional partitions share one hardware unit;
d. Mapping a generic functional description into an implementation on a particular set of components, either as software suitable for a given microprocessor or logic which can be implemented from the given hardware libraries.

An embedded system is an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through two ways that computational processes interact with the physical world: reaction to a physical environment and execution on a physical platform. Common *reaction* constraints specify deadlines, throughput and jitter and originate from behavioral requirements. Common *execution* constraints bound available processor speeds, power, and hardware failure rates and originate from implementation choices. Control theory deals with reaction constraints; computer engineering deals with execution constraints. The key to embedded systems design is gaining control of the interplay between computation and both kinds of constraints to meet a given set of requirements on a given implementation platform.

EMBEDDED PROCESSORS AND SOFTWARE ARCHITECTURES

Before we can begin to architect an embedded system or its firmware, we must have clear requirements. Properly written requirements define the WHAT of a product. WHAT does the product do for the user, specifically? For example, if the product is a ventilator, the list of WHAT it does may include a statement such as: "If power is lost during operation, the ventilator shall resume operation according to its last programmed settings within 250ms of power up. Each requirement statement must also be two other things: unambiguous and testable .Testability is key. If a requirement is written properly, a set of tests can be easily constructed to

verify that requirement is met. Decoupling the tests from the particulars of the implementation, in this manner, is of critical importance. Any coupling between the test and the implementation is problematic. The design of the software architecture-the division of the function into communicating processes-is closely related to automobile engine design [4]. Two systems with identical functions but different process structures may run at very different speeds, require vastly different amounts of memory, etc.

The increase in the number and complexity of electronic devices in vehicles is affecting the way designers conceive the entire automobile. For example, a high-range car equipped with a rich set of options can have over 100 wires tied to a dashboard (including 30 to 35 wires entering each front door), for a total length of about 3 miles and a weight of about 100 pounds (Mercedes S-series and Renault Safrane)[8]. Effectively replacing such a bundle of hard-wired devices is a local area network implemented, usually, with a serial bus**.** This bus replaces the numerous wires, allocating the functions to CPUs that no longer need be topologically close to the controlled part, such as a window or mirror. Compensating for the added cost of the electronics are savings in material (wires and connectors), manufacturing time, and reliability of the links. The complexity of these functions and the precision dictated, for example, by the exhaust emission control laws, require the use of specialized fast microcontrollers. Such controllers, like the Motorola MC68332 feature 32-bit architectures (versus the 8 or 16 bits of older microcontrollers) along with specialized micro-programmable processors (time processing units) to handle hard real-time tasks more easily. Fig 1.0 shows a typical sequence of steps in a top down design of an embedded system; of course, most actual design processes will mix top-down and bottom-up design.

Circuit tunability for energy/performance-scalable operation is another issue. As CMOS technology scaling has continued to increase transistor density, new challenges have emerged in the design of logic circuits, on-chip interconnect, off-chip interconnect, and memory hierarchy, all of which are further complicated by increased device variability in shrinking CMOS process technologies and reduced supply voltages. The limitations of logic were the first to affect the traditional progression of processor development when transistor density caused CMOS designs to hit a thermal density barrier. Energy efficiency can be improved significantly by scaling the supply voltage (nominally around 1V) to sub/near-threshold levels (0.5–0.6 V), at the expense of a significant exponential increase in delay. This energy/throughput scalability exists not only for on-chip computation, but also for on-chip/off-chip inter connect. The most common solution to the thermal density issues has been to increase computational parallelism and decrease supply voltages and clock frequencies. This increased parallelism, in turn, exhibits larger demands on communications, both within and between these on-chip cores and processor sockets. This is especially true in the high-performance computing environment, where multicore processors require both large on-chip as well as off-chip memory bandwidth. These individual, multicore systems must communicate with each other via high-speed, off-chip interconnect between cores, chips, boards, racks, and rooms. The result is that the energy required for communication, either on-chip or off-chip, can outweigh the energy required for the logic to actually perform a computation.

TRENDS

Language- and synthesis-based origins

The first generation of methodologies traced their origins to one of two sources: *Language-based* methods lie in the software tradition, and *synthesis-based* methods stem from the hardware tradition. A language based approach is centered on a particular programming language with a particular target runtime System (often fixed-priority scheduling with preemption). Early examples include Ada and, more recent, RT-Java. Synthesis-based approaches have evolved from circuit design methodologies. They start from a system description in a tractable, often structural, fragment of a hardware description language such as VHDL and Verilog and automatically derive an implementation that obeys a given set of constraints.

Implementation platform independence

The second generation of methodologies introduced a semantic separation of the design level from the implementation level to gain maximum independence from a specific execution platform during early design phases. There are several examples. The synchronous programming languages embody an abstract hardware semantics (synchronicity) within software; implementation technologies are available for different platforms, including bare machines and time-triggered architectures. System C combines a synchronous hardware semantics with asynchronous execution mechanisms from software (C++); implementations and require partitioning into components that will be realized in hardware on the one side and in software on the other. The semantics of common dataflow languages such as Mat-lab's Simulink are defined through a simulation engine, as is the controller specification in Figure 1; implementations focus on generating efficient code. Languages for describing distributed systems, such as the Specification and Description Language (SDL), generally adopt an asynchronous semantics.

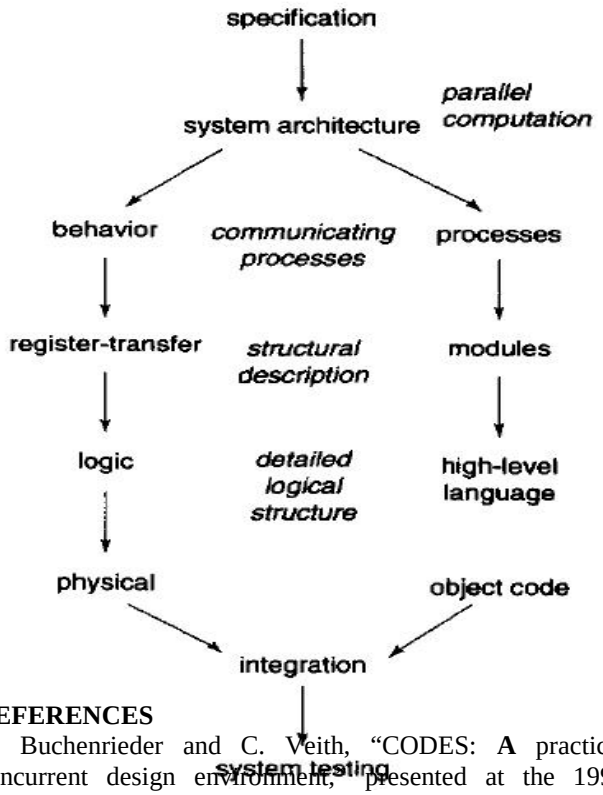
Execution Semantics Independence

The third generation of methodologies is based on modeling languages such as the Unified Modeling Language (UML) and Architecture Analysis and Design Language (AADL) and go a step beyond implementation independence. They attempt to be generic not only in the choice of implementation platform, but even in the choice of execution and interaction semantics for abstract system descriptions. This leads to independence from a particular programming language, as well as to an emphasis on system architecture as a means of organizing computation, communication, and resource constraints. Much recent attention has focused on frameworks for expressing different models of computation and their interoperation. These frameworks support the construction of systems from components and high-level primitives for their

coordination. They aim to offer not just a disjoint union of models within a common Meta Language but also to preserve properties during model composition and to support meaningful analyses and transformations across heterogeneous model boundaries.

CONCLUSION

High-performance embedded systems consist of multiple HW/SW subsystems, with application software tasks distributed over heterogeneous processor subsystems using sophisticated interconnects .The HW/SW interface and the CPU subsystems must handle the interaction between software tasks and the interconnect structure. The interface provides the application software layer with an abstraction of the SoC architecture, called a *parallel programming model*. It also includes a network interface for both multiprocessor booting and inter-processor communication that connects the subsystem to the network. When the SoC includes more than one CPU, HW/SW interface design becomes more complicated. Parallel programming models are more complex than uniprocessor programming models; similarly, network interfaces are more complex than a unified memory. Thus, as a recent multiprocessor SoC case study confirms, the HW/SW[15] interface could become a key challenge in heterogeneous SoC design. Because design teams traditionally have applied a software or hardware-only strategy, there is a temptation to continue using this approach to implement large applications. Software teams claim that their approach results in a shorter design cycle. For example, a pure software approach may reduce the design cycle for derivative design because software is flexible enough to add new functionality. On the other hand, hardware teams argue that their approach is more efficient. While an embedded software approach could result in a larger chip or even a chipset, the ASIC approach will yield a smaller chip. Even for a single product, achieving the best volume in a given market window considering chip size and yield in chip production may require combining hardware and software solutions. In terms of yield in chip production, both ASIC and embedded software approaches have pros and cons. The ASIC approach can suffer from low yield in the first few months of chip production until the learning curve improves. However, the reduced chip size may improve total chip production. An embedded software approach can give a good initial yield since it reuses an already proven SoC platform. However, a larger chip size may reduce the effects of yield improvement. Ultimately, achieving optimal SoC production will require some combination of hardware and software solutions. This co-design scheme opens the design process to several optimizations that are not possible using the classic approach in which hardware and software are designed separately. The most obvious improvement is better adaptation of the CPU to both hardware and software interfaces. For example, designers can use new flexible processor technologies such as Tensilica7 to optimize performance at the HW/SW interface by introducing application-specific I/O operation[2]. In addition, using reconfigurable hardware, such as the Xilinx Virtex II Pro, can optimize hardware interfaces to an embedded CPU.



REFERENCES

K. Buchenrieder and C. Veith, "CODES: **A** practical concurrent design environment," presented at the 1992 ACM/IEEE Int. Workshop on Hardware-Software CO-Design, Estes Park CO,Oct. 1993.

M. Chiodo and A. **Sangiovanni-Vincentelli,** "Design methods for reactive real-time systems co-design," presented at the 1992 ACMDEEE Int. Workshop on Hardware-Software CO-Design. Estes Park CO, Oct. 199j.

M. Chiodo, P. Giusto. A. Jurecska. M. Marelli. L. Lavagno. H. Hsieh, and A. **Sangiovanni-Vincentelli,** "A formal specifiation model for hardware/software co-design," presented at the Int. Workshop on Hardware Software CO-Design, Cambridge MA, Oct. 1993.

J. G. D'Ambrosio, **S.** Hu, and A. Tang, "The role of analysis in hardware Software co-design," presented at the 1993 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Cambridge MA. Oct. 1993.

R. Emst, J. Henkel, and Th. Benner, "Hardware-software co-synthesis for micro-controllers," **IEEE Des.** & **Test of Comput.,** vol. 9, pp. 138-153, 1990 -vol. 10, no. 4, pp. 64-75, Dec. 1993.

R. K. Gupta and G. De Micheli, "Hardware-software co-synthesis for digital systems," **IEEE Des.** & **Test Comput.,** vol. **10,** no. 3, pp. 2941, Sept. 1993

D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R.Sherman, A. Shtull-Trauring, and M. Trakhtenbrot,

STATEMATE: A working environment for the development of complex reactive systems," **IEEE Trans. Software Eng.,** vol. 16, no. 4, pp. 403414, Apr. 1990.

A. Kalavade and E. A. Lee, "A hardware-software co-design methodology for DSP applications," **IEEE Des.** & **Test,** vol. 10, no. 3, pp. 1628, Sept. 1993.

H. Kasahara and **S.** Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," **IEEE Trans. Comput.,** vol. C-33, no. 11, pp. 1023-1029, Nov. 1984.

S. Malik and A. Wolfe, "Tutorial on embedded systems performance analysis," presented at ICCD'93, Cambridge MA, Oct.1993.

P. Chou, R. Ortega, and G. Borriello, "Synthesis of the hardware software interface in microcontroller-based systems," in **Proc. ICCAD-92.** IEEE Computer Society Press, 1992, pp.488495.

P. Michel, U. Lauther, and P. Duzy, Eds., **The Synthesis Approach to Digital System Design.** Norwell, MA: Kluwer,1992.

M. Burke, "An interval-based approach to exhaustive and incremental inter-procedural data-flow analysis," **ACM Trans. Programming Languages and Systems,** vol. 12, no 3, pp. 341-395, July, 1990.

T. L. Adam, K. M. Chandy. and J. R. Dickson, "A comparison of list schedules for parallel processing systems," **Commun. ACM,** vol. 17, no. 2, pp. 685-690, Dec. 1974.

A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," **Proc. IEEE,** vol. 79, no. 9, pp.' 127&1282, Sept. 1991.

W. W. Chu and L. M.-T. Tan, "Task allocation and precedence relations for distributed real-time systems," **IEEE Trans. Comput.,** vol. C-36, no. 6, pp. 667-679, June 1987.