Computer Aided Embedded Systems Design

Ireneusz Karkowski
Delft University of Technology
The Netherlands

Abstract

This paper gives an overview of the existing approaches to the hardware/software codesign of embedded systems. We discuss all the steps in the design trajectory. The common approach of adding dedicated hardware to accelerate the software (by way of a FPGA or ASIC coprocessor) is judged as not worth the effort. The multi-processor framework of ASIPs (application specific instruction set processors), where instead of coprocessor special functional units are added to the processor core, is identified as a much more interesting basis for embedded system design.

1 Introduction

Embedded systems are usually defined as combinations of hardware with general purpose computational capabilities and more dedicated modules. Together they perform a function carefully partitioned in software and hardware to obtain the optimum trade-offs between the various performance metrics. These types of implementations have become increasingly popular as advances in ic-technology and processor architectures allow for flexible computational parts and high-performance modules integrated on a single carrier. An example embedded system architecture is presented in figure 1.

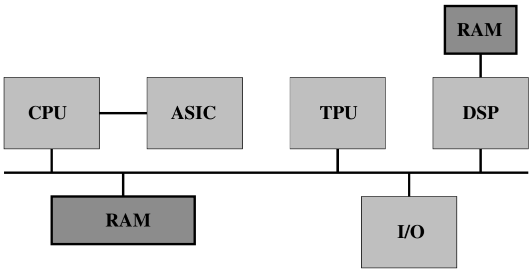

Figure 1: Example embedded system. TPU is a *time procesing unit*, for example from Motorola.

Unfortunately, it is nowadays still common practice, that all system level decisions are taken ad hoc in the beginning of the design process. Months of efforts are then invested in realizing them, often manually hand-crafting the embedded software in assembly language. Since usually there is no time left for the second try, many of these decisions have to be conservative to guarantee the system correctness. Also the design reuse is limited, which results in longer time-to-market times. Clearly, this leads to less market competitive products.

The design automation for embedded systems is becoming a "must". What is needed is an interactive environment that supports the designer in transforming an algorithmic specification into a suitable implementation. The environment should allow fast experimentation with different architectural options and release the designer from the burden of many time consuming, but often simple, synthesis tasks. Fortunately the research efforts of the last years have brought several interesting results making implementation of the environments for embedded system design feasible. Several such prototype environments have been realized. We can classify them according to the nature of the embedded systems that they support:

- Data dominated or computation oriented systems. In this kind of systems dedicated hardware is added mostly to "accelerate" the computation. A typical example in this category is **COSYMA** [16].

- Control dominated, reactive systems. Dedicated hardware is used for the concurrency exploitation and/or to meet hard real-time constraints. Examples in this class include **VULCAN** [17] or **Chinook** [11].

Many embedded application, especially in mobile communication, are a collection of both control and data dominated subtasks. Therefore, the latest research efforts are directed towards supporting mixed systems [5].

The existing environments differ significantly in many aspects. For example in the choice of the application specification language, target architecture selection and/or hardware/software partitioning algorithms. Clearly however the general design flow inside these environments exhibits several similarities. A typical view on the hardware/software codesign trajectory is presented in figure 2. In the following sections we will concentrate on the most important aspects of the embedded system design and discuss how design automation addresses them.

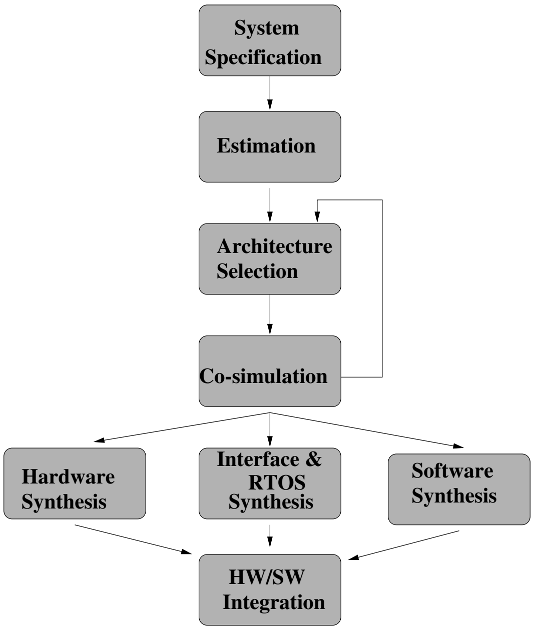

Figure 2: Hardware/software codesign trajectory

2 System Specification

Construction of any codesign environment is only possible if the embedded system specification is given in a high level language capturing the expected behavior of the system. This specification should have the following properties:

1. Large expressiveness, allowing description of the software and hardware behavior, including its low-level, real-time and concurrency aspects.

2. Could serve as input to both, the existing software compilation systems, and to hardware synthesis tools for the hardware parts.

3. Hide obscure implementation details, which could hamper the re-targetability.

The existing environments encompass a broad range of different application domains, and therefore also description styles. The simplest, and simultaneously most popular, approach involves using **ANSI C** (or **C++**) programming language for the software part specification and a **Verilog** hardware description language (HDL) for the hardware parts [32, 16]. Use of both languages is necessary since **ANSI C** is a rather bad HDL. This approach has several disadvantages. If the hardware/software partitioning is not given a priori, this may require that the designer provides double specifications for every part of the application. Another deficiency lies in the lack of the process concurrency and real-time support in **ANSI C**. This becomes a serious problem in case of reactive (control-dominated) embedded systems. Use of well known multi-processing programming languages, as for example **Ada**, has been completely disregarded because of unacceptably high run-time overhead. Instead several augmented versions of **ANSI C** have been introduced, with **C^x** of **COSYMA** [16] or **HardwareC** of **VULCAN** [17] as perfect examples. To avoid the deficiencies of the **ANSI C/Verilog** approach many researchers used a single description language, which is both more suitable for concurrent systems' specifications, and can be used as input to both hardware and software generation subsystems. As example, **TTL** (Templated T-LOTOS) [8] has formal syntax and semantics and can be converted to both **ANSI C** and **RTL** (Register Transfer Language). Other examples involve: **Occam II** [19] in TOSCA system [3] (extended to include timing constraints among events), **Esterel** [6], **CSP** [19] and Co-design Finite State Machines (**CFSM**) [9] in POLIS. Recently **Java** [28] received much attention as a specification language for embedded systems [10]. The language was originally intended for embedded applications, has built-in multi-threading, timing control, synchronization and exception handling support. Even more, **Java** represents a new approach to re-targetability throught its virtual machine and standardized programmers API. Unfortunately several problems have to be solved before **Java** becomes a broadly accepted specification language for real-time embedded systems: mainly its implementation inefficiency (interpretation and garbage collection) and limited real-time support. One may conclude that *the* solution to the problem of system specification does not (yet) exist.

3 Hardware and Software Estimation

Obtaining good hardware/software cost and performance estimates is essential before any design space exploration can be started. While different approaches are possible, usually the estimation process takes place before the architecture selection stage. Software estimates (i.e. #clock cycles, code size, execution rate) can be obtained relatively easily by source code profiling. Several dedicated profilers are available. Among them the **CINDERELLA** performance analysis tool by Sharad Malik [26] or **KUKLFLAP** program execution time estimator [30] by Paul Stravers. The estimation has to be performed for all candidate CPUs.

The latency, area and power consumption of the application specific hardware is usually not known if the unit has not been synthesized yet. The parameters can be estimated for example using methodology from [34]. Often also standard hardware components are used. The cost/performance information about them can be then retrieved from the manufacturer's data books. In most cases however data about software and hardware is still imprecise. The way to handle imprecision during hardware/software codesign is proposed in [22]; it relies on so called *possibilistic programming*.

4 Architecture Selection

Under architecture selection we understand here solving two subproblems:

- Selecting a collection of processing components to constitute the system and deciding on a way they interact (*allocation*).
- Mapping the application into these components (*binding and scheduling*), or in other words performing hardware/software partitioning.

The second problem received most attention in the past. Most hardware/software partitioning systems assume that the *allocation* is already given. Usually it includes a general purpose processor (*software*) communicating with an application specific coprocessor (*hardware*), implemented as ASIC or FPGA. The application's specification is partitioned between *software* and *hardware* to meet the timing constraints and in the same time minimize the overall system cost. The following optimization techniques were, among the others, applied to solve the problem: simulated annealing [16], hierarchical clustering [36], dynamic programming [24], integer programming [22] and a modified Karnighan/Lin heuristic [33]. The last method is reported to have best quality/computational complexity properties. It is however not so flexible as simulated annealing or integer programming approaches in taking different design constraints and cost functions. Recently also several methods have been proposed with attempt to tackle *allocation* together with the *binding* and *scheduling* problems. For this purpose integer programming [4], simulated annealing, genetic algorithm, tabu search [2] and evolutionary algorithm [31] have been applied.

5 Co-simulation

Once the system's architecture has been selected it has to be simulated to verify its behavior against the specification. Besides functional also timing verification is necessary (using hardware simulators). Simulation engines for embedded systems should be capable of handling heterogeneous and interacting components, and therefore their design and implementation forms quite a challenge. The most commonly used approach involves using the **PTOLEMY** system [7] for that purpose. Because of the enormous complexity of the embedded systems, some systems allow simulation at different levels of detail, from cycle-accurate to behavioral (Honeywell **PML** [25]). In addition to simulation, use of some description languages (for example **CFSM** [9]) allows formal verification of the specification. If the simulation shows that system does not satisfy the requirements, we go back to the architecture selection stage.

6 Hardware and Software Synthesis

Only when the result of the co-simulation is positive the synthesis stage can begin. Hardware tasks can be synthesized by feeding their description in an HDL into one of the existing hardware compilers (for example from **Synopsys** [23]). Software can also usually be compiled using a standard compiler delivered for the CPU of the system. Most work and expertise requires implementation of the hardware/software interfaces. Few subsystems have been designed to completely automate or at least significantly ease this task. As example we can give here the **CoWare** environment from IMEC [35]. Some researchers go even further and attempt to combine interface generation with the real-time operating system synthesis, as in SERRA system from Stanford University [27].

7 Discussion

The described above existing environments for embedded systems have already been successfully applied to several real-time designs. Often they are not only portable but also easily re-targetable. In general however most of the environments have difficulties in handling large diversity of the embedded systems' architectures and applications. Also a fundamental problem arises: is adding a dedicated coprocessor to your system the right way of accelerating the software? Looking at the the Amdahl law [1] one must conclude, that adding the coprocessor to your system is only worth the effort if we can find a part of application which represents at least 90% of the total computation time [15]; otherwise the overall speedup is neglectable or even negative. In practical embedded applications such large piece of code seldom can be identified, and even then it is impossible or inefficient to implement this dedicated hardware as an FPGA or ASIC, due to the necessity of implementing shared memory access mechanism.

A multiprocessor embedded system based on ASIP (application specific instruction set processor, like MOVE [12]) seems to be a much better alternative. Especially, if the system consists of several (different) ASIP processors, each with functionality optimized for the subtasks which they have to perform. These processors should be automatically generated and easily tailored to your application using dedicated design automation frameworks, as the existing **MOVE framework** [14]. Code partitioning among the processors leads then to exploitation of the course-grain parallelism (task parallelism and data parallelism in loops [21]), while the fine-grain (instruction level) parallelism [20] is exploited locally by each of the processors. Also so called *special functional units* (SFUs) implementing application specific operations (for example **fft**) can be easily added to each processor. This leads us to the generic multi-processor architecture for codesing presented in figure 3.

This solution does not suffer from the communication overhead because SFUs have, in contrast to coprocessors, access to the same register set as regular FUs. SFUs can also, although to a lesser extend, be used for serial code parts. E.g., an experiment using the MCCD (minimum cost contour detection) algorithm, which draws contours within a

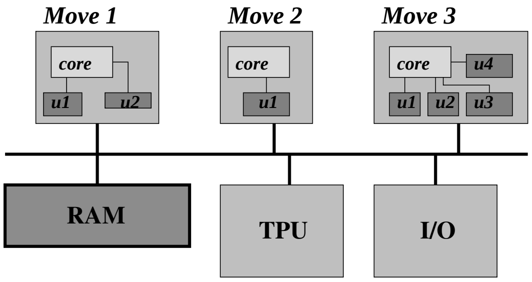

Figure 3: MOVE based multiprocessor architecture for codesign. Block with names starting with u are SFUs.

graphical image (see [13]), showed that a specific routine could be accelerated with a factor of two by adding a five input adder. Occasionally this adder might also be used within the other code to do regular, two input, additions.

It may also occur that, as soon as we have the capability of adding FUs (because of the scalability requirement), it is more efficient, and easier, to add more regular FUs instead of a SFU, especially when the ASIP has to support an application domain instead of just one application; the reasons are well known to people designing RISC processors [29, 18]. In this case the serial code part profits even more from these extra FUs.

To gain some insight, the performance of a processor with an extra SFU is modeled and compared to a GPP (general purpose processor) with a coprocessor. The coprocessor approach is discussed first. Assume a fraction f of a program, called the parallel part, is down-loaded to and executed by the coprocessor; the execution time of this fraction decreases with a speedup factor S_{par}. The other fraction $1-f$, the serial part, is executed on the GPP. The execution time for a processor with coprocessor configuration is then estimated by:

$$t_{coproc} = (1-f) \times t_{proc} + \frac{f \times t_{proc}}{S_{par}} + t_{comm}$$

where t_{proc} is the execution time of the program on the same GPP without a coprocessor, and t_{comm} is the communication time required between processor and coprocessor. The latter is modeled to be proportional to execution of the number of instructions within the down-loaded code part on the processor. It is given by:

$$t_{comm} = rct \times f \times t_{proc}$$

where rct (the relative communication time) is a proportionality constant.

If a SFU is incorporated within the processor we assume again that the same speedup of the parallel fraction. In this case the serial part also shows a little speedup S_{serial}. There is no communication overhead. The execution time is then given by:

$$t_{SFU} = \frac{(1-f) \times t_{proc}}{S_{serial}} + \frac{f \times t_{proc}}{S_{par}}$$

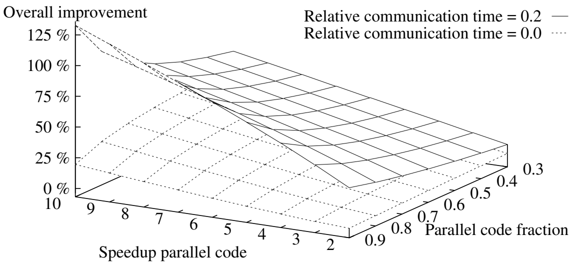

Figure 4: Performance improvement of a processor with SFU compared to one with coprocessor.

In figure 4 the performance improvement when using a SFU instead of a coprocessor is drawn as a function of S_{par} and f for two rct values: one with zero communication overhead, $rct = 0$, the other with $rct = 0.2$. The improvement of the serial code fraction, in case a SFU is used, is pessimistically taken to be 1/20 of the improvement of the parallel code fraction, i.e. $S_{serial} = 1 + (S_{par} - 1)/20$. The figure shows that, given the assumptions, a SFU easily outperforms a coprocessor, even when we neglect the coprocessor communication overhead.

8 Conclusions

In this paper we gave an overview of the existing approaches to the hardware/software codesign of embedded systems. We stated that the existing environments can be of great help to the designers in typical application domains. In our opinion however, in the past, too much attention has been put on the kinds of architectures, which usually deliver neglectable gains in comparison with "all software" solutions. The common approach of adding dedicated coprocessor hardware to accelerate the software (in form of FPGA or ASIC coprocessor) has been judged not worth the effort. We proposed the multi-processor framework for ASIP generation as a much better basis for embedded system design, because application specific functionality is implemented as SFUs, instead of a coprocessor.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Atlantic City, New Jersey, April 1967.

[2] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[3] A. Balboni, W. Fornaciari, and D. Sciuto. Co-synthesis and co-simulation of control dominated embedded systems. *Journal of Design Automation for Embedded Systems*, 1(3):257–289, July 1996.

[4] A. Bender. Design of an optimal loosely coupled heterogeneous multiprocessor system. In *Proceedings of The European Design & Test Conference*, 1996.

[5] T. Benner and R. Ernst. An approach to mixed systems codesign. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[6] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), 87-152 1991.

[7] J. T. Buck, S. Ha, E. A. Lee, and Messerschmitt D. G. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994. special issue on Simulation Software Development.

[8] V. Carchilo, M. Malgeri, and G. Mangioni. TTL: A LOTOS extension for system description. In *Proceedings of Basys'96*, Lisboa(Portugal), 1996.

[9] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software codesign of embedded systems. *IEEE micro*, 14(4):26–36, August 1994.

[10] P. Chou and G. Borriello. Software architecture synthesis for retargetable real-time embedded systems. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[11] P. Chou, R.B. Ortega, and G. Borriello. The chinook hardware/software co-synthesis system. In *International Symposium on System Synthesis*, Cannes, France, September 1995.

[12] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[13] Henk Corporaal and Jan Hoogerbrugge. Code generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, pages 240–259. Kluwer Academic Publishers, 1995. Chapter 14.

[14] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[15] M. Edwards. Software acceleration using coprocessors: Is it worth the effort? In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[16] R. Ernst and J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. In *International Workshop on Hardware-Software Codesign*, Boston, September 1992.

[17] R.K. Gupta, C.N. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of DAC'92*, pages 225–230, 1992.

[18] John L. Hennessy and David A. Patterson. *COMPUTER ARCHITECTURE, a Quantitative Approach, Second Edition*. Morgan Kaufmann publishers, 1995.

[19] C.A. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Engelwood Cliffs, NJ, 1985.

[20] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996.

[21] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[22] I. Karkowski and R.H.J.M. Otten. An Automatic Hardware-Software Partitioner Based on the Possibilistic Programming. In *Proceedings of the ED&TC Conference*, Paris, March 1996.

[23] D. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Prentice-Hall, 1996.

[24] P. Knudsen and J. Madsen. Pace: A dynamic programming algorithm for hardware/software partitioning. In *International Workshop on Hardware-Software Codesign*, 1996.

[25] S. Kumar and F. Rose. A codesign environment supporting hardware/software modeling at different levels of detail. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[26] Y.-T. S. Li and S. Malik. Performance analisys of embedded software using implicit path enumeration. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[27] V. Mooney, T. Sakamoto, and G. De Micheli. Run-time scheduler synthesis for hardware-software systems and application to robot control design. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[28] K. Nilsen. Issues in the Design and Implementation of Real-Time Java. *WWW document*, http://www.new-monics.-com/webroot/technologies/java/RTJI.ps, 1996.

[29] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33, October 1980.

[30] Paul Stravers. KUKLFLAP program execution time estimator. *Gesellschaft f. Mathematik u. Datenverarbeitung, Sankt Augustin, Germany*, http://alcatraz.gmd.de:-9422/castle/doc/kuklflap.html, 1995.

[31] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[32] M. Theißinger, P. Stravers, and H. Veit. Castle: An interactive environment for hw-sw co-design. In *Proc. of the HW-SW Co-Design Conf.*, 1994.

[33] F. Vahid. Modifying min-cut for hardware and software functional partitioning. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[34] F. Vahid and D.D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on VLSI Systems*, 3(6):459–464, September 1995.

[35] K. van Rompaey, D. Verkest, I. Bolsens, and H. De Man. CoWare - A design environment for heterogeneous hardware/software systems. In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1996.

[36] X.Xiong, E. Barros, and W. Rosentiel. A method for partitioning UNITY language in hardware and software. In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.