SYSTEM-LEVEL DESIGN TECHNIQUES FOR ENERGY-EFFICIENT EMBEDDED SYSTEMS

System-Level Design Techniques for Energy-Efficient Embedded Systems

by

MARCUS T. SCHMITZ
University of Southampton, United Kingdom

BASHIR M. AL-HASHIMI
University of Southampton, United Kingdom

and

PETRU ELES
Linköping University, Sweden

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-48736-5
Print ISBN: 1-4020-7750-5

To our beloved families

Contents

List of Figures

List of Tables

Preface

It is likely that the demand for embedded computing systems with low energy dissipation will continue to increase. This book is concerned with the development and validation of techniques that allow an effective automated design of energy-efficient embedded systems. Special emphasis is placed upon system-level co-synthesis techniques for systems that contain dynamic voltage scalable processors which can trade off between performance and power consumption during run-time.

The first part of the book addresses energy minimisation of distributed embedded systems through dynamic voltage scaling (DVS). A new voltage selection technique for single-mode systems based on a novel energy-gradient scaling strategy is presented. This technique exploits system idle and slack time to reduce the power consumption, taking into account the individual task power dissipation. Numerous benchmark experiments validate the quality of the proposed technique in terms of energy reduction and computational complexity.

The second part of the book focuses on the development of genetic algorithm-based co-synthesis techniques (mapping and scheduling) for single-mode systems that have been specifically developed for an effective utilisation of the voltage scaling approach introduced in the first part. The schedule optimisation improves the execution order of system activities not only towards performance, but also towards a high exploitation of voltage scaling to achieve energy savings. The mapping optimisation targets the distribution of system activities across the system components to further improve the utilisation of DVS, while satisfying hardware area constraints. Extensive experiments including a real-life optical flow detection algorithm are conducted, and it is shown that the proposed co-synthesis techniques can lead to high energy savings with moderate computational overhead.

The third part of this book concentrates on energy minimisation of emerging distributed embedded systems that accommodate several different applications within a single device, i.e., multi-mode embedded systems. A new co-synthesis technique for multi-mode embedded systems based on a novel operational-mode-state-machine specification is presented. The technique increases significantly the energy savings by considering the mode execution probabilities that yields better resource sharing opportunities.

The fourth part of the book addresses dynamic voltage scaling in the context of applications that expose extensive control flow. These applications are modelled through conditional task graphs that capture control flow as well as data flow. A quasi static scheduling technique is introduced, which guarantees the fulfilment of imposed deadlines, while at the same time, reduces the energy dissipation of the system through dynamic voltage scaling.

The new co-synthesis and voltage scaling techniques have been incorporated into the prototype co-synthesis tool LOPOCOS (Low Power Co-Synthesis). The capability of LOPOCOS in efficiently exploring the architectural design space is demonstrated through a system-level design of a realistic smart phone example that integrates a GSM cellular phone transcoder, an MP3 decoder, as well as a JPEG image encoder and decoder.

Acknowledgments

Financial support of the work was provided by the Department of Electronics and Computer Science at the University of Southampton, the Embedded Systems Laboratory (ESLAB) at Linköping University, as well as the Engineering and Physical Sciences Research Council (EPSRC), UK.

Special thanks go to the members of the Electronic Systems Design Group (ESD) at the University of Southampton, for many fruitful discussions.

We would like to thank Christian Schmitz, who has contributed in deriving the smart phone benchmark during a visit at the University of Southampton.

We would also like to acknowledge Neal K. Bambha (University of Maryland, USA) and Flavius Gruian (Lund University, Sweden) for kindly providing their benchmark sets, which have been used to conduct some of the presented experimental results.

Chapter 1

INTRODUCTION

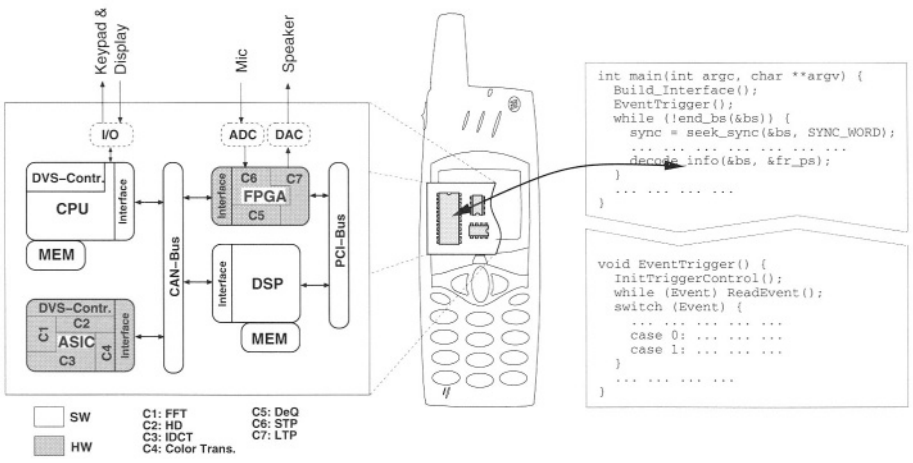
Over the last several years, the popularity of portable applications has explosively increased. Millions of people use battery-powered mobile phones, digital cameras, MP3 players, and personal digital assistants (PDAs). To perform major pans of the system's functionality, these mass products rely, to a great extent, on sophisticated embedded computing systems with *high performance* and *low power dissipation.* The complexity of such devices, caused by an ever-increasing demand for functionality and feature richness, has made the design of modern embedded systems a time-consuming and error-prone task. To be commercially successful in a highly competitive market segment with tight time-to-market and cost constraints, computer-based systems in mobile applications should be cheap and quick to realise, while, at the same time, consume only a small amount of electrical power, in order to extend the battery-lifetime. Designing such embedded systems is a challenging task.

This book addresses this problem by providing techniques and algorithms for the automated design of energy-efficient distributed embedded systems which have the potential to overcome traditional design techniques that neglect important *energy management* issues. In this context, special attention is drawn to *dynamic voltage scaling* (DVS) — an energy management technique. The main idea behind DVS is to dynamically scale the supply voltage and operational frequency of digital circuits during run-time, in accordance to the temporal performance requirements of the application. Thereby, the energy dissipation of the circuit can be reduced by adjusting the system performance to an appropriate level. Furthermore, the proposed synthesis techniques target the coordinated design (co-design) of *mixed hardware/software* applications towards the effective exploitation of DVS, in order to achieve substantial reductions in energy.

The main aims of this chapter are to introduce the fundamental problems that are involved in designing distributed embedded systems and to provide the terminology used throughout this work. The remainder of this chapter is organised as follows. Section 1.1 outlines a typical system-level design process. A task graph specification model, used to capture the system's functionality, is introduced in Section 1.2. Section 1.3 describes the individual system design steps using some illustrative examples. Hardware and software synthesis are briefly discussed in Section 1.4. Finally, Section 1.5 gives an overview of the book contents.

1.1 Embedded System Design Flow

A typical embedded system, as it can be found, for example, in a smart-phone, is shown in Figure 1.1. It consists of heterogeneous components such



```
int main(int argc, char **argv) {
  Build_Interface();
  EventTrigger();
  while (!end_bs(&bs)) {
    sync = seek_sync(&bs, SYNC_WORD);
    ... ... ... ... ... ... ...
    decode_info(&bs, &fr_ps);
  }
  ... ... ... ...
}

void EventTrigger() {
  InitTriggerControl();
  while (Event) ReadEvent();
  switch (Event) {
    ... ... ... ... ...
    case 0: ... ... ...
    case 1: ... ... ...
  }
  ... ... ... ...
}
```

(a) Embedded architecture: A distributed heterogeneous system

(b) Embedded software

Figure 1.1. Example of a typical embedded system (smart-phone)

as software programmable processors (CPUs, DSPs) and hardware blocks (FP-GAs, ASICs). These components are interconnected through communication links and form a distributed architecture, such as the one shown in Figure 1.1 (a). Analogue-to-digital converters (ADC), digital-to-analogue converters (DAC), as well as input/output ports (I/O) allow the interaction with the environment. A complete embedded system, however, consists additionally of application software (Figure 1.1 (b)) that is executed on the underlying hardware architecture (Figure 1.1(a)). Clearly, effective embedded system design demands optimisation in *both* hardware and software parts of the application. When designing an embedded computing system, as part of a new product, it is common to go through several design steps that bring a novel product idea down to its physical realisation. This is usually referred to as system-level design flow. A possible and common design flow is introduced in Figure 1.2. It is characterised by three important design steps: *system specification* (Step A), *co-synthesis* (Step B), as well as concurrent *hardware and software synthesis* (Step C). The remainder of this section briefly outlines this design flow.

Starting from a new product idea, the first step towards a final realisation is *system specification.* At this stage, the functionality of the system is captured using different conceptual models [61] such as natural language, annotated-graphic representations (finite state machines, data-flow graphs), or high-level languages (VHDL, C/C++, SystemC). This design step is indicated as Step A in Figure 1.2. Having specified the system's functionality, the next stage in the design flow is the *co-synthesis,* shown as Step B in Figure 1.2. The goal of co-synthesis is threefold:

Architecture allocation: Firstly, an adequate target architecture needs to be allocated, i.e., it is necessary to determine the quantity and the types of different interconnected components that form the distributed embedded system. Components that can be allocated are given in a predefined technology library.

Application mapping: Secondly, all parts of the system specification have to be distributed among the allocated components, that is, tasks (function fragments) and communications (data transfers between tasks) are uniquely mapped to processing elements and communication links, respectively.

Activity scheduling: Thirdly, a correct execution order of tasks and communications has to be determined, i.e., the activities have to be scheduled under the consideration of interdependencies.

These three co-synthesis stages aim to optimise the design according to objectives set by the designer, such as power consumption, performance, and cost. In order to reduce the power consumption, emerging co-synthesis approaches (as the one proposed in this work) tightly integrate the consideration of *energy management techniques* within the design process [67, 76, 99, 100].

Energy management Energy management techniques utilise existing idle times to reduce the power consumption by either shutting down the idle components or by reducing the performance of the components.

The consideration of energy management techniques during the co-synthesis allows the optimisation of allocation, mapping, and scheduling towards their effective exploitation. After the co-synthesis has allocated an architecture as well as mapped and scheduled the system activities (tasks and communications), the next stage in the design flow is the concurrent *hardware and software synthesis,* indicated as Step C in Figure 1.2. These separated design steps transform the system specification, which has been split between hardware and software, into

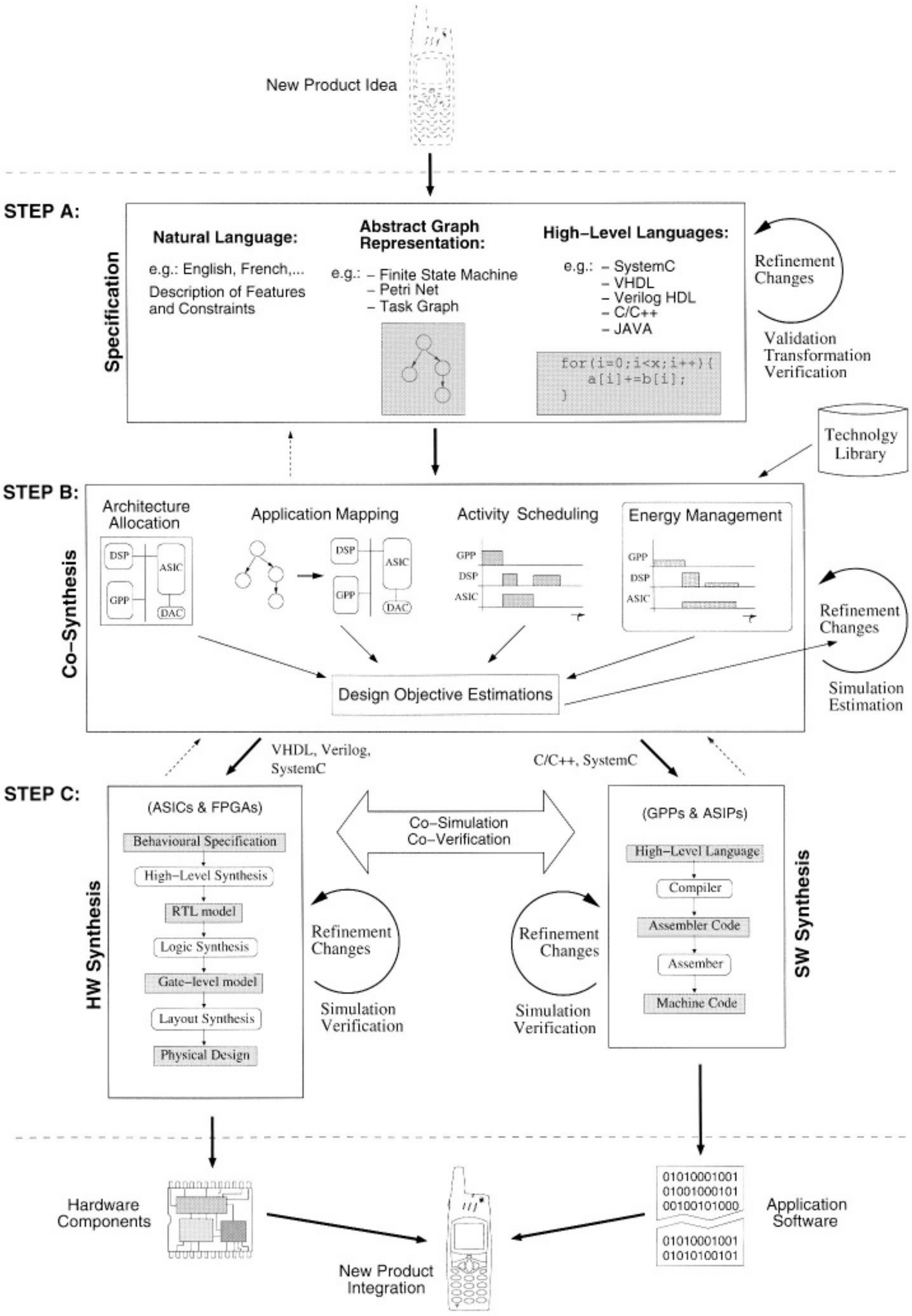


Figure 1.2. Typical design flow of a new embedded computing system

physical implementations. System parts that are mapped onto customised hardware are designed using high-level [8, 19, 60, 134, 154], logic [9, 42, 110, 131], and layout [56] synthesis tools. While system parts that have been mapped onto software programmable processors (CPUs, DSPs) are compiled into assembler and machine code, using either standard or specialised compilers and assemblers [1, 93]. The main advantage of a concurrent hardware (HW) and software (SW) synthesis is the possibility to co-simulate both system parts, with the aim of finding errors in the design as early as possible to avoid expensive re-designs. The following section describes the whole design process shown in Figure 1.2 in more detail and introduces the terminology used throughout this book.

1.2 System Specification (Step A)

The functionality of a system can be captured using a variety of conceptual specification models [61]. Different modelling styles are, for example, high-level languages (hardware description and programming languages) such as SystemC, Verilog HDL, VHDL, C/C++, or JAVA, as well as more abstract models such as block diagrams, task graphs, finite state machines (FSMs), Petri nets, or control/dataflow graphs. Typical applications targeted by the presented work can be found in the audio and video processing domain (e.g. multi-media and communication devices with extensive data stream operations). Such applications fall into the category of data-flow dominated systems. An appropriate representation for these systems is the task graph model [84, 112, 157], which will be introduced in the following section.

1.2.1 Task Graph Representation

The functionality of a complex system with intensive data stream operations can be abstracted as a directed, acyclic graph (DAG) $G_S = (\mathcal{T}, \mathcal{C})$, where the set of nodes $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_n\}$ denotes the set of tasks to be executed, and the set of directed edges $\mathcal{C}$ refers to communications between tasks, with $\gamma_{ij} \in \mathcal{C}$ indicating a communication from task τ_i to task τ_j. A task can only start its execution after all its ingoing communications have finished. Each task can be annotated with a deadline θ, the time by which its execution has to be finished. Furthermore, the task graph inherits a repetition period ϕ which specifies the maximal delay between to invocations of the source tasks (tasks with no ingoing edges). Structurally, task graphs are similar to the data-flow graphs that are commonly used in high-level synthesis [60, 154]. However, while nodes in data-flow graphs represent single operations, such as multiplications and additions, the nodes in task graphs are associated with larger (coarse) fragments of functionality, such as whole functions and processes. The concept behind this model can be exemplified using a simple illustrative example.

Example 1: For the purpose of this example, consider an MP3 audio decoder. In order to reconstruct the "original" stereo audio signal from an encoded stream, the decoder reads the data stream and applies several transformations such as Huffman decoding, dequantisation, inverse discrete cosine transformation (IDCT), and antialiasing. A possible task graph specification along with a high-level language description in C of such an MP3 decoder is shown in Figure 1.3. The figure outlines the relation between task graph model and high-level description. In this particular example the granularity of each task in the task graph corresponds to a single sub-function of the C specification. For instance, the `Huffman Decoder` tasks (τ_3 and τ_4) in Figure 1.3(a) reflect the functionality that is performed by the third sub-function in Figure 1.3(b). The flow of data is expressed by edges between the individual tasks. The output data produced by the `Huffman Decoder` tasks, for example, is the input of the `dequant` tasks (τ_5 and τ_6), indicated by the communication edges $\gamma_{3,5}$ and $\gamma_{4,6}$. In order to decode the compressed data into a high quality audio signal, one execution of all tasks in the graph, starting from task τ_0 and finishing with τ_{16}, has to be performed in at most $25ms$ as expressed by the task deadline θ_{16}. However, to obtain real-time decompression of a continuous music stream, the execution of all tasks has to be performed 40 times per second, i.e., with a repetition rate of $\phi = 25ms$. Although in this particular example the deadline and the repetition rate are identical, they might vary in other applications. As opposed to the C specification, the task graph explicitly exhibits *application parallelism* as well as *communication between tasks (data flow),* while the exact algorithmic implementation of each function is abstracted away. ❑

Task graphs can be derived from given high-level specification either manually or using extraction tools, such as the one proposed in [148].

1.3 Co-Synthesis (Step B)

Once the system's functionality has been specified as task graph, the system designers will start with the system-level co-synthesis. This is indicated as Step B in Figure 1.2. In addition, Figure 1.4 shows the co-synthesis flow in diagrammatic form. Co-synthesis is the process of deriving a mixed hardware/software implementation from an abstract functional specification of an embedded system. To achieve this goal, the co-synthesis needs to address four fundamental design problems: *architecture allocation, application mapping, activity scheduling,* and *energy management.* Figure 1.4 shows the order in which these problems have to be solved. In general, these co-synthesis steps are iteratively repeated until all design constraints and objectives are satisfied [52, 54, 70, 156]. An iterative design process has the advantage that valuable feedback can be provided to the different synthesis steps. This feedback, which