Application Scenarios in Streaming-Oriented Embedded System Design

Stefan Valentin Gheorghita, *Student Member, IEEE,* Twan Basten, *Senior Member, IEEE,* and Henk Corporaal

(Invited Paper, SOC 2006 Best Paper)

***Abstract*—In the past decade, real-time embedded systems became more and more complex and pervasive. From the user perspective, these systems have stringent requirements regarding size, performance and energy consumption, and due to business competition, their time-to-market is a crucial factor. Therefore, much work has been done in developing design methodologies for embedded systems to cope with these tight requirements. This paper presents an overview of the basic steps of a design method for handling the increasing dynamism in modern embedded system applications. It uses the concept of *application scenarios* that group operation modes of an application that are similar from the resource usage perspective and describes how to incorporate them in the overall real-time embedded system design process. We furthermore briefly describe our automated scenario-based design trajectory for reducing the energy consumption of a streaming application running on a single processor platform via dynamic voltage and frequency scaling. Two case studies show the use of application scenarios for low energy design, under both soft and hard real-time constraints.**

***Index Terms*—Embedded Systems, Design Methodology, Application Scenarios, Dynamic Voltage and Frequency Scaling.**

I. Introduction

EMBEDDED systems usually consist of processors that execute domain-specific applications. Much of their functionality is implemented in software, which is running on one or multiple processors, leaving only the high performance functions implemented in hardware. Typical examples of embedded systems include TV sets, cellular phones, MP3 players, smart cameras and wireless access points. Most of these systems are running multimedia and/or telecom applications, which typically exhibit dynamic behavior, and of which execution costs (e.g., number of cycles, energy) depend on the input data. Moreover, these applications are often implemented as a main loop, called the loop of interest, that is executed over and over again, reading, processing and writing out individual stream objects (see figure 1). A stream object might be a bit belonging to a compressed bitstream representing a coded video clip, a macro-block, a video frame, an audio sample, or a network package. Usually, these applications have to deliver a given throughput (number of objects per second), which imposes a time constraint on each loop iteration.

The read part of the loop of interest takes a stream object from the input stream and separates it into a *header* and the object's *data*. The processing part consists of several kernels. The write part sends the processed data to output devices, like a screen or speakers, and saves the internal state of the application for further use (e.g., in a video decoder, the previous decoded frame may be necessary for decoding the current frame). The dynamism existing in modern applications leads to the usage of different kernels for each stream object, depending on the object type. The actions executed in a particular loop iteration form an internal *operation mode* of the application.

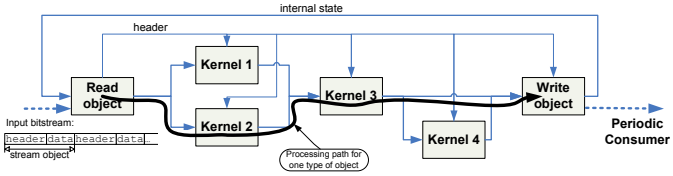

Fig. 1. Typical streaming application processing a stream object.

This paper presents an overview of a method that aims to provide a systematic way of detecting and exploiting both at design time and runtime a characteristic of the application that has not been fully used in embedded system design previously, namely the different internal operation modes. The approach combines the static analysis and profiling of a system, that is done at design time, with information collected at runtime about the environment in which the system is used. If possible operation modes in which a system may run, together with information about their resource consumption, are known at design time, specific and aggressive design decisions can be made for each operation mode in different design steps. To avoid complexity problems, the operation modes that are closely related to each other from a resource consumption perspective are clustered in *application scenarios*, distinguishing the operation modes that are really different. The result is a faster or lower energy implementation (e.g., by using different source code optimizations per scenario), or a better estimation of required resources (e.g., the number of computation cycles or bandwidth) may be derived, leading to *a smaller, cheaper and more energy efficient system that can deliver the required performance*.

The paper is organized as follows. Section II illustrates the difference between application scenarios and the well known use-case scenarios. Section III presents the role of application scenarios in an embedded system design flow and provides examples of scenario exploitation found in the literature. A classification of application scenarios is given in section IV. An MP3 case study is detailed throughout the paper, with section V presenting the implemented design trajectory and the obtained energy savings under both hard and soft real-time constraints. Section V also summarizes the numerical

All authors are with the Electrical Engineering Department, Electronic Systems Group, Eindhoven University of Technology, The Netherlands, e-mail: {s.v.gheorghita,a.a.basten,h.corporaal}@tue.nl.

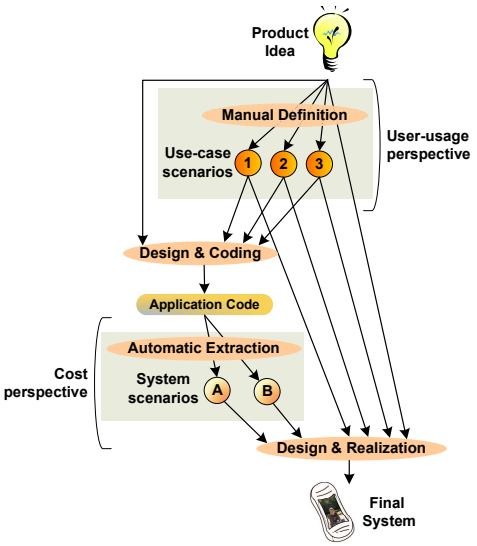

Fig. 2. Full design flow.

results obtained for a motion compensation video kernel. Some conclusions are discussed in the last section.

II. Use-case vs. Application Scenarios

Scenario-based design has been in use for a long time in different areas [1], like human-computer interaction or object oriented software engineering. In these cases, scenarios concretely describe, in an early phase of the development process, the use of a future system. Moreover, they appear like narrative descriptions of envisioned usage episodes, or like unified modeling language (UML) use-case diagrams that enumerate, from a functional and timing point of view, all possible user actions and system reactions that are required to meet a proposed system functionality. These scenarios are called *use-case scenarios*, and characterize the system from the *user perspective*. In the embedded systems area, they were used in both hardware [2] and software design [3].

In this paper, we concentrate on a different type of scenarios, so-called *application scenarios*, which may be derived from the behavior of the system. These scenarios are used to reduce the system cost by exploiting information about what can happen at run-time to make better design decisions. While use-case scenarios classify the application's behavior based on the different ways it can be used, application scenarios classify it from the *resource usage perspective*. This type of scenarios was first identified and exploited by researchers from IMEC, Belgium, in [4].

Definition: *An application scenario is a detectable set of operation modes of an application that are sufficiently similar in a multi-dimensional resource-based cost space (e.g., execution cycles, memory usage, source code).*

The cost space is defined over the dimensions of interest for a specific problem. For example, we might be interested in operation modes that share the same source code, or that execute in the same number of CPU cycles. To be exploited, these modes must be detectable in the application, preferably as soon as the application starts to execute in one of them. As the definition is very general, it has to be tailored to the specific design problem at hand (e.g., the application behavior for a specific type of input data [5]).

Figure 2 depicts a design trajectory using use-case and application scenarios. It starts from a product idea, for which the stakeholders[1] define the product's functionality as use-case scenarios. These scenarios characterize the system from a user perspective and are used as an input to a design process that includes both software and hardware components. In order to optimize the design of the system, the detection and use of application scenarios augment this trajectory (the bottom gray box in the figure). Once the application is coded, its scenarios related to resource utilization are extracted in a semi-automatic way, and they are considered for the decisions made during the following phases of the system design. The sets of use-case scenarios and application scenarios are not necessarily disjoint. One or more use-case scenarios may be merged in one application scenario, a use-case scenario may be split into several application scenarios, or several application scenarios may intersect several use-case scenarios.

Case study: We want to design a portable MP3 player as a USB stick. At first sight, there are two main use-case scenarios: (i) the player is connected to the computer and music files are transferred between them, and (ii) the player is used to listen to music. These scenarios can be divided in more detailed use-case scenarios, like, for the second one, song selection, play or fast forward scenarios. Let us consider the play scenario. From the software point of view, this use-case can be split into two different application scenarios: (i) mono mode and (ii) stereo mode. Exploiting these scenarios, the system battery lifetime may be increased, because mono mode requires less compute power. Thus a lower supply voltage may be used, while still meeting the timing constraints of the decoding.

III. Application Scenario Methodology

The methodology for introducing application scenarios into the current embedded system design trajectory consists of three steps depicted in figure 3. These are influenced by the design context in which they are applied, but nevertheless they are still quite general:

- *Identification:* given the application, how is it classified into scenarios?
- *Predictor/detector derivation:* given a particular iteration of the loop of interest, to which scenario does it belong?
- *Exploitation:* given a particular scenario, what can be done to optimize the system when it executes in this scenario?

1: *Scenario identification* starts from the original application and identifies its different operation modes. Their resource usage (in the cost space of interest) is characterized, and the modes with similar needs are clustered in an application scenario. The methods used for scenario identification can be

[1]The stakeholders are persons, entities, or organizations who have a direct stake in the final system; they can be owners, regulators, developers, users or maintainers of the system.

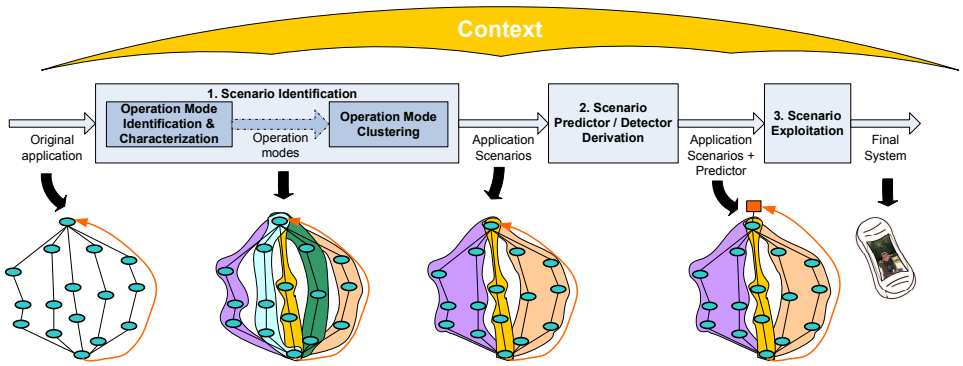

Fig. 3. A scenario-based design flow for embedded systems.

divided into three categories: (i) analytical, (ii) profiling and (iii) hybrid. Independent of the method, the set of the identified application scenarios must cover all possible application operation modes.

In an *analytical method*, the application structure is statically analyzed to identify similar operation modes. This method is restrictive, as it can not automatically collect information about how the application is really used and how it behaves at runtime (e.g., which is the most frequently used scenario at runtime). The real runtime behavior of the application can be captured using a *profiling method*, but in this case it is more difficult to derive scenario predictors than it is in the analytical case, and in general not all scenarios may be identified as not all possible distinct operation modes may be covered by profiling. To overcome this problem, an extra scenario, called the *backup scenario*, must be considered. It is selected at runtime when the application is running in an operation mode that did not appear during profiling. A *hybrid method* combines the advantages of the previous two methods and it is the most powerful way to determine scenarios.

Especially for profiling and hybrid methods, an explosion in the number of operation modes may occur during their identification. In this case, decisions must be made using partial information, applying the operation mode identification and clustering simultaneously. A trade-off should be made between how many different scenarios and operation modes may be handled during the identification process and how accurate the identification is. The optimal number of scenarios depends on many factors, including their intended usage.

Besides the bottom-up approach that first identifies a large set of operation modes of the application, and afterwards tries to combine them based on similarity, there is also a top-down approach. This initially considers the application as a single scenario, and then, recursively splits each scenario based on the differences between the operation modes it covers. The two approaches are not fundamentally different, being just different ways to implement scenario identification.

Case study: The context of our case study is to reduce the energy consumption of a sequential streaming application by exploiting the dynamic voltage and frequency scaling feature of modern processors. In case of hard real-time constraints, we use a static analysis for operation mode and scenario identification, augmented with a profiling approach for selecting which are the most common scenarios that appear during the system usage. In case of soft real-time, profiling is the basis for identification, as it can give us the most accurate view of the system utilization. However, for collecting internal information about the application (e.g., application structure or variables), we combine it with the static analysis approach.

2: Runtime scenario *detector* and/or *predictor* derivation is the step of finding a way to determine in which scenario the application runs at a certain moment in time. The current scenario of an application can be either *detected*, based on already known information (e.g., variable values), or it can be *predicted*, with a certain confidence. Detection can be seen as prediction with 100% confidence. At runtime, when a change in the scenario in which the application runs is predicted, a switch is done by the system and the optimizations applied for this specific scenario are activated.

Different ways of implementing predictors may be considered, like static vs. runtime adaptive or centralized vs. distributed. Independent of the predictor implementation, the following information may be used: (i) runtime application internal information like variable values and executed code (i.e., a basic block that appears only in one scenario); (ii) statistical information obtained by profiling or from the application designer (e.g., how often a scenario may appear at runtime); (iii) a probabilistic scenario transition model, like a Markov chain; and (iv) a history of active scenarios in the current execution. Predictors may be of two types:

- *Reactive*: Only information already computed by the application is used.
- *Proactive*: A part of the application control-flow that follows the predictor is duplicated/extracted in the predictor source code. This allows early decision making. There is a trade-off between the amount of code duplicated and how early in the execution the current application scenario can be predicted. Usually, the earlier the better, but the prediction overhead must be limited.

Case study: Under both soft and hard real-time constraints, we derive reactive centralized predictors, using the application variables that do not change their values for one entire iteration of the loop of interest. The place where the predictor may be inserted into the application source code depends on the considered variables and where in the loop body assignments to these variables occur. In case of hard real-time constraints,

the predictor was extracted using a static analysis, so it is in fact a static detector. For soft real-time constraints, the information collected via profiling was used to derive the predictor. To guarantee a required system quality, we use a calibration mechanism (discussed in section V-B2) that helps the predictor to learn on the fly, which makes it a runtime adaptive predictor.

3: *Scenario exploitation* is the step that uses scenarios to optimize a design, by applying more aggressive optimizations for each scenario. As this step strongly depends on what the designer wants to achieve, we survey several papers that use application scenarios (often without explicitly defining or identifying the concept).

The application scenario concept was identified explicitly for the first time in [4], where it was used to improve the mapping of dynamic applications onto a multiprocessor platform. Concepts closely related to the scenario idea already appear in [6]. In other work, the concept was applied several times in an ad-hoc manner, with an emphasis on the exploitation of scenarios, mostly ignoring the (automatic) identification and prediction. The work in [7] concentrates on saving energy for a sequential application. The targeted architecture has a single processor with reconfigurable components (e.g., number and type of function units), and its supply voltage can be changed. For each manually identified scenario, the most energy efficient architecture configuration that still meets the timing constraints is selected. It is not clear how scenarios are predicted at runtime. In [8], a reactive predictor is used to select the lowest supply voltage for which the timing constraints of an MPEG decoder are met. An extension [9] considers two simultaneous resources for scenario characterization. It looks for the most energy efficient configuration for encoding video on a mobile platform, exploring the trade-off between computation and compression efficiency.

Recently, scenarios have also been used in the context of the geometrical loop transformation framework to extend the scope of the applicability of the geometrical model [10]. The work combines profiling with the geometrical model to find the optimal scenarios for global memory optimizations. The idea is similar to hyperblock scheduling, but on a much coarser level. The authors have a systematic way of detecting operation modes based on profiling and of clustering them in scenarios based on a trade-off between the number of memory accesses and the code size increase.

In the context of multi-task applications, besides the already mentioned work of [4], the use of application scenarios for reducing the energy consumed by a multi-task application mapped on a voltage scaling aware processor is also investigated in [11]. Other work in the multi-task context is [12]. The considered scenarios are characterized by different communication requirements (e.g., bandwidth, latency) and traffic patterns. The paper presents a method to map application communication to a network on chip architecture, satisfying the design constraints of each individual scenario.

Scenarios were also used to improve the operating system. In [13], the authors present a way of optimizing dynamic memory allocation (i.e., MALLOC()/FREE()) for the IPv4 layer in an IEEE 802.11b wireless network application. Different allocation algorithms are used for different scenarios, which are identified based on the possible network package sizes.

All mentioned papers except [10] emphasize scenario exploitation for a given application, and not for a class of applications, and do not go into detail on identification and prediction, which are important topics in our work.

IV. Application Scenario Classification

The different classes of embedded systems (e.g., hard vs. soft real-time) and the design problem that is optimized lead to multiple possible criteria that can be used for scenario classification.

Considering how scenario switches are driven at runtime, two main scenario categories can be considered: *data flow driven* and *event driven*. *Data flow driven scenarios* characterize different executed actions in an application that are selected at runtime based on the input data characteristics (e.g., the type of streaming object). Usually each scenario has its own implementation within the application source code. *Event driven scenarios* are selected at runtime based on events external to the application, such as user requests or system status changes (e.g., battery level). They typically characterize different quality levels for the same functionality, which may be implemented as different algorithms or different quality parameters in the same algorithm. They are also called *quality scenarios*. The two types of scenarios may form a hierarchy. For different quality levels, a data flow driven scenario may require different amounts of resources for the same application source code.

The runtime switches that occur between scenarios are differentiated by the *tolerable amount of side-effects*. Usually, in case of data flow driven scenarios, side-effects are not acceptable, whereas in case of event driven scenarios different potential side-effects may be acceptable. For example, a switch between quality scenarios in a TV set may appear as an image format change (e.g., from 4:3 to 16:9), with an acceptable side-effect of image flickering during system reconfiguration. But when the TV tuner switches from a data driven scenario to another one, no side-effects that visibly affect the image are acceptable.

As design methods for single and multi-task systems concentrate on different aspects, scenarios can also be classified in *intra-task scenarios*, which appear within a sequential part of an application (i.e., a task), and *inter-task scenarios* for multi-task applications. This classification can also be seen as a hierarchy. Usually, the scenario in which a multi-task application is running is derived from the scenarios in which each application task is currently running. Data flow driven intra- and inter-task scenarios are conceptually the same from the resource usage and runtime switching perspectives, but they have a different impact on the intra- and inter-task parts of the design flow, and their exploitation is in general different.

Finally, scenario usage differs for *soft* and *hard real-time systems*. Not all methods presented above for each step of the methodology can always be applied. For example, for hard real-time systems, scenario identification can only use static analysis, and only detectors may be used to identify the

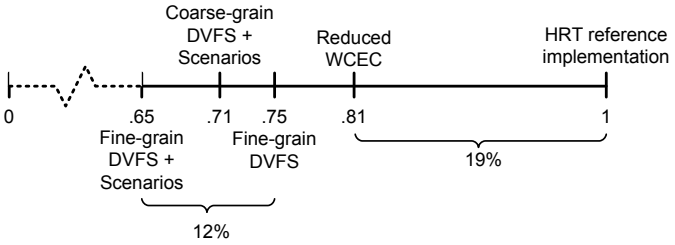

Fig. 4. Normalized energy for different MP3 hard real-time implementations

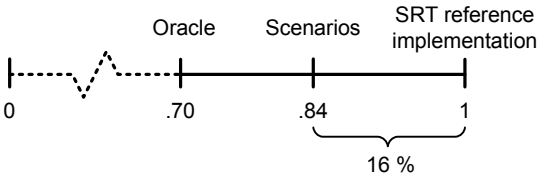

Fig. 5. Normalized energy for different MP3 soft real-time implementations

current scenario at runtime, whereas for soft real-time systems predictors and statistical information from profilers may be used.

Case study: For the MP3 application, we concentrate on intra-task data driven scenarios, under both soft and hard real-time constraints. As each scenario characterizes the decoding of an audio sample type, and a stream contains different sample types, we take care that there are no side effects during switching.

V. Our Trajectory and Experimental Results

As the presented methodology is quite general, this section presents results for an automatic trajectory we developed for a specific design problem, namely the identification, prediction and exploitation of application scenarios to reduce the energy consumed by a single task streaming application running on a dynamic voltage and frequency scaling (DVFS) aware processor. DVFS is an effective energy-saving technique that consists of varying the frequency and voltage of a processor at runtime according to processing needs. Most of the algorithms integrated in our tools are general and they may be used for different problems. Our trajectory works for both hard and soft real-time constraints. It starts from an application written in C, as C is the most used language to write embedded systems software, and generates the final energy-aware implementation also in C.

A. Experimental Setup

The numerical results presented below refer to energy consumption of an Intel XScale PXA255 processor, measured using the XTREM simulator [14]. We consider that the processor frequency (f_{CLK}) can be set discretely within the operational range of the processor, with 1MHz steps. The supply voltage (V_{DD}) is adapted accordingly, using the following equation:

$$f_{CLK} = k \cdot \frac{(V_{DD} - V_T)^2}{V_{DD}}, \qquad (1)$$

where $V_T = 0.3V$ and constant $k = 208.3$ is computed for $V_{DD} = 1.5V$ and $f_{CLK} = 200$MHz. A frequency/voltage transition overhead $t_{switch} = 70\mu s$ was considered, during which the processor stops running. The energy consumed during this transition is $4\mu J$. When the processor is not used, it switches to an idle state within one cycle, and it consumes an idle power of 63mW. This situation occurs if the decoding of a streaming object needs less computation cycles than estimated.

B. MP3 Decoder

For the evaluation we considered a benchmark consisting of a randomly selected set of 20 stereo and 10 mono streams, as stereo songs are more often listened to than mono songs.

1) Hard Real-Time: Our trajectory may generate different energy saving implementations, from a purely static one to an implementation that uses a fine-grain DVFS-aware scheduler. A comparison is shown in figure 4 and discussed below.

As hard real-time systems require a conservative design approach, in [15], we describe a method and tool that use a static analysis of the application source code to identify and exploit scenarios for reducing the over-estimation in worst case number of execution cycles (WCEC) compared to existing methods. The scenarios incorporate correlations that differentiate between the source code parts that never and the ones that may execute together in the same iteration of the loop of interest. To avoid an explosion in the number of detected scenarios, the correlations are extracted using only information about the automatically detected application parameters with a large impact on the execution time.

For the MP3 decoder, the estimated WCEC for the entire application, which is the maximum between the ones obtained for each scenario, was reduced with 9.3%. As for hard real-time systems no deadlines may be missed, the processor must be able to execute at least the WCEC per decoding time period for an audio sample. Thus, the processor frequency can be lowered with 9.3%, saving 19% in energy consumption when decoding our audio stream benchmark.

By exploiting the different WCEC for each scenario, the energy can be further reduced. Our trajectory may generate a proactive predictor that acts like a DVFS-aware coarse-grain scheduler that selects once per loop iteration the processor frequency and the supply voltage level. For our benchmark, the energy reduction is 29%, compared to the reference implementation.

Our trajectory can also augment scenarios in a fine-grain scheduler which changes the processor frequency multiple times during an iteration of the loop of interest. The combination of scenarios with a state-of-the-art fine-grain DVFS-aware scheduler for hard real-time systems, described in [5], reduces the average energy with 12% compared to using only the fine-grain DVFS-aware scheduler, and with 35% compared to the original reference. Fine-grain DVFS gives better results than coarse-grain DVFS if the frequency switching time is small enough relative to period of the application loop of interest. For larger switching times, fine-grain DVFS is infeasible or coarse-grain DVFS outperforms it.

2) Soft Real-Time: For soft real-time systems, a certain deadline miss ratio is acceptable, so using a conservative

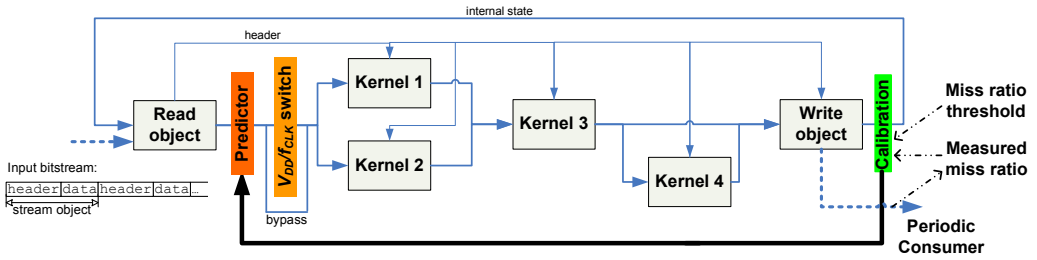

Fig. 6. Final implementation for a soft real-time application.

trajectory leads to an over-dimensioned system. Hence, we refined our trajectory to consider also information collected by profiling the application. In [16], we describe a method and a tool-flow that can automatically detect the most important application parameters and use them to define and dynamically predict scenarios in the context of soft real-time systems. The trajectory inserts into the application source code three components (see figure 6): a predictor, a switching mechanism and a runtime calibration mechanism. The third component does not exist in the hard real-time case, and it is a mechanism that tunes the predictor to keep the miss ratio under a given threshold. This mechanism is necessary as it is difficult, or almost impossible, to cover during profiling all possible operation modes in which a system may run. It helps the predictor to learn at runtime about newly discovered operation modes. Using the generated code, the average energy consumed by the MP3 decoder is reduced with 16% compared to the reference without scenarios, for a measured miss ratio of never more than one audio sample per 6 minutes (0.008%). Comparing with the maximum energy reduction that can be obtained (the oracle in figure 5), we have reached about 50% of the theoretically possible reduction. We plan to further reduce energy consumption by developing a more intelligent calibration mechanism.

C. Motion Compensation Video Kernel

For this kernel, we did the same experiments as for the MP3 decoder. Under hard real-time constraints, we saved up to 69% using a coarse-grain scheduler and up to 75% when using a fine-grain scheduler, when comparing with the reference implementation. In the soft real-time context, the savings were up to 16%, which represents about 50% of the theoretically possible reduction.

VI. Conclusions

This paper presented the basic steps of a methodology that exploits the concept of application scenarios in embedded system design. Application scenarios cluster the operation modes of a system that are similar from the resource usage perspective, allowing for more aggressive design decisions per scenario, thus complementing the well known use-case scenarios. We described our scenario-based design trajectory for reducing the energy consumption of a sequential streaming application under both hard and soft real-time constraints. We showed that for an MP3 decoder and a motion compensation video kernel substantial energy savings may be obtained compared to state-of-the-art methods. Our trajectory exploits the difference in the required computation cycles between different scenarios. It can be easily adapted to consider another resource (e.g., the number of memory accesses or memory size). For more details about scenario-based design, the interested reader is referred to http://www.es.ele.tue.nl/scenarios.

Acknowledgment

This work was supported by the Dutch Science Foundation, NWO, project FAME, number 612.064.101.

References

[1] J. M. Carroll, Ed., *Scenario-based design: envisioning work and technology in system development*. John Wiley & Sons Inc, 1995.

[2] J. Paul, D. Thomas, and A. Bobrek, "Scenario-oriented design for single-chip heterogeneous multiprocessors," *IEEE Transations on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 868–880, 2006.

[3] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*. Addison Wesley Publishing Company, 2004.

[4] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *Proc. 15th Int. Symposium on System Synthesis (ISSS)*. ACM, 2002, pp. 112–119.

[5] S. V. Gheorghita, T. Basten, and H. Corporaal, "Intra-task scenario-aware voltage scheduling," in *Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, 2005, pp. 177–184.

[6] P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, and H. De Man, "Dynamic memory oriented transformations in the MPEG4 IM1-Player on a low power platform," in *Proc. of the 1st International Workshop on Power-Aware Computer Systems*. Springer-Verlag, London, UK, 2000, pp. 40–50.

[7] R. Sasanka, C. J. Hughes, and S. V. Adve, "Joint local and global hardware adaptations for energy," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 144–155, 2002.

[8] M. Pedram, W. Cheng, K. Dantu, and K. Choi, "Frame-based dynamic voltage and frequency scaling for a MPEG decoder," in *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, USA, 2002, pp. 732–737.

[9] D. G. Sachs, S. V. Adve, and D. L. Jones, "Cross-layer adaptive video coding to reduce energy on general-purpose processors," in *Proc. of IEEE Int. Conf. on Image Processing*, 2003, pp. 109–112.

[10] M. Palkovic, H. Corporaal, and F. Catthoor, "Global memory optimisation for embedded systems allowed by code duplication," in *Proc. of the ACM Workshop on Software and Compilers for Embedded Systems*, 2005, pp. 72–79.

[11] S. Lee, S. Yoo, and K. Choi, "An intra-task dynamic voltage scaling method for SoC design with hierarchical FSM and synchronous dataflow model," in *Proc. of the Int. Symp. on Low Power Electronics and Design*. ACM, 2002, pp. 84–87.

[12] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *Proc. of Design, Automation and Test in Europe (DATE)*. IEEE, 2006, pp. 1–6.

[13] S. Mamagkakis, D. Soudris, and F. Catthoor, "Middleware design optimization of wireless protocols based on the exploitation of dynamic input patterns," in *Proc. of Design, Automation, and Test in Europe (DATE)*. IEEE, 2007, pp. 118–123.

[14] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Y. Lueh, "XTREM: A power simulator for the Intel XScale core," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 115–125, July 2004.

[15] S. V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal, "Automatic scenario detection for improved WCET estimation," in *Proc. of the 42nd Design Automation Conf. (DAC)*. ACM, 2005, pp. 101–104.

[16] S. V. Gheorghita, T. Basten, and H. Corporaal, "Scenario selection and prediction for dvs-aware scheduling," *Journal of VLSI Signal Processing*, Accepted for publication, http://dx.doi.org/10.1007/s11265-007-0086-1.