Application Scenarios in Streaming-Oriented Embedded System Design

Stefan Valentin Gheorghita, Twan Basten and Henk Corporaal
EE Department, ES Group, Eindhoven University of Technology, The Netherlands
{s.v.gheorghita,a.a.basten,h.corporaal}@tue.nl

***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. In this paper, we introduce the concept of *application scenarios* that group operation modes of an application that are similar from the resource usage perspective, and we describe how to incorporate them in the overall real-time embedded system design process. A case study shows the use of application scenarios for low energy design, under both soft and hard real-time constraints.**

I. Introduction

Embedded systems usually consist of processors that execute domain-specific programs. Much of their functionality is implemented in software, which is running on one or multiple generic processors, leaving only the high performance functions implemented in hardware. Typical examples include TV sets, cellular phones, MP3 players and printers. Most of these systems are running multimedia and/or telecom applications, like video and audio decoders. These applications are usually 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, or an audio sample. 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. For each stream object some of these kernels are used, depending on the object type. 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 actions executed in a loop iteration form an internal *operation mode* of the application.

In this work, we introduce ways of detecting and exploiting a characteristic of the application that has not been fully used in embedded system design previously, namely the different internal operation modes, each with their own typical resource consumption. Operation modes that are closely related to each other from a resource consumption perspective are clustered in so-called application scenarios, distinguishing operation modes that are really different. If these scenarios are considered in different steps of the embedded system design, a

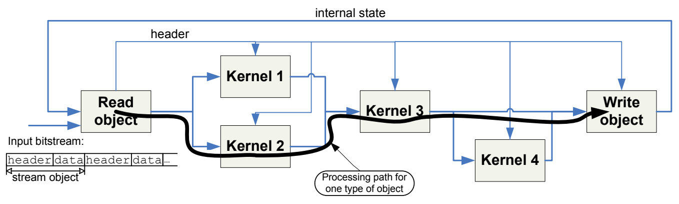


Fig. 1. Typical streaming application processing a stream object.

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. These intermediate results lead to *a smaller, cheaper and more energy efficient system that can deliver the required performance*.

The paper is organized as follows. Section II presents the role of application scenarios in an embedded system design flow, illustrating the difference between them and the well known use-case scenarios, and it provides examples of scenario exploitation found in the literature. A classification of application scenarios is given in section III. A case study showing how we reduced the energy consumption of a single task system under both hard and soft real-time constraints is presented in section IV. Some conclusions are discussed in the last section.

II. Scenarios in Design

A. Use-case vs. Application Scenarios

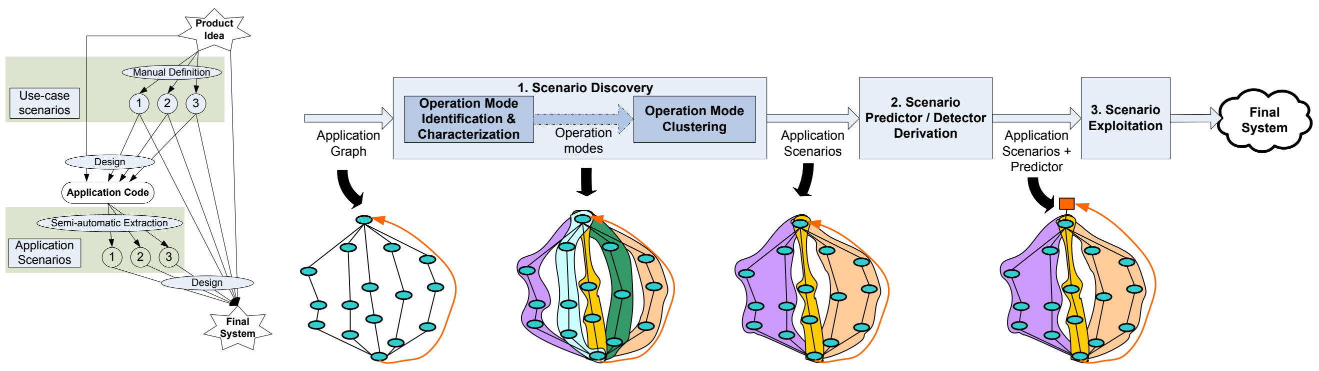
Scenario-based design has been in use for a long time in different areas [1], [2], like human-computer interaction or object oriented software engineering. In these cases, scenarios concretely describe, in an early phase of the development process, the use of a future system. Moreover, they appear like 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 [3], [4] and software design [5].

In this work, we concentrate on a different kind of scenarios, so-called *application scenarios*, that characterize the system from the *resource usage perspective*.

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

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



(a) Full design flow (b) Application scenario usage methodology

Fig. 2. A scenario based design flow for embedded systems.

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 is commonly tailored to the specific design problem at hand (e.g. the application behavior for a specific type of input data [6]).

Figure 2(a) depicts a design flow using scenarios. It starts from a product idea, for which the stakeholders define the future utilization as use-case scenarios. They are used in a user-centric development process to design an embedded system that includes both software and hardware components. In order to optimize the design of the system, we suggest to augment this trajectory with application scenarios (the bottom gray box in figure 2(a)). Once the application is coded, its scenarios related to resources 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.
Example: 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 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 in two different application scenarios: (i) mono mode and (ii) stereo mode. If these scenarios are used during the design, the system battery lifetime may be increased, as in case of playing in mono mode a lower computation power is needed, thus a lower supply voltage may be used to meet the timing constraints of the decoding.

B. Application Scenario Usage Methodology

The methodology to introduce application scenarios into the current embedded system design trajectory consists of three steps depicted in figure 2(b): (1) *discovery*, (2) *predictor/detector derivation* and (3) *exploitation*.

1: *Scenario discovery* 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 discovery can be divided in 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 discovered 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 discover scenarios.

Especially for profiling and hybrid methods, an explosion in the number of operation modes may appear during their identification. In this case, decisions must be made using partial information, applying mode identification and clustering simultaneously. There is a trade-off between how many different scenarios and modes may be handled during the discovery process (from which the process speed and memory usage are derived) and the resulting quality. Discovery and clustering may be performed in a bottom-up approach, but also in a top-down refinement based 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.

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.

3: *Scenario exploitation* is the step that uses scenarios to optimize a design. As this step strongly depends on what the designer wants to achieve, we present an overview of several papers that use application scenarios (although in general they do not give an explicit definition and/or identify the concept).

In [7], the authors concentrate on saving energy for a single task application. For each manually identified scenario they select the most energy efficient architecture configuration that can be used to meet the timing constraints. The architecture has a single processor with reconfigurable components (e.g. number and type of function units), and its supply voltage can be changed. 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 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.

To reduce the number of memory accesses, in [10], the authors selectively duplicate parts of application source code, enabling global loop transformations across data dependent conditions. They 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. The final application implementation, including scenarios and the predictor, is done manually.

In context of multi-task applications, the scenario concept was first used in [11] to capture the data-dependent dynamic behavior inside a thread, to better schedule a multi-thread application on a heterogenous multi-processor architecture, allowing the change of voltage level for each individual processor. 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 [12]. Other work in the multi-task context is [13]. 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.

Most of the mentioned papers (except [10]) emphasize scenario exploitation and do not go into detail on discovery and prediction. Our work focuses on these last two problems.

III. Application Scenario Classification

The different classes of embedded systems (e.g. hard vs. soft real-time) and the problem that must be solved 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 actions executed 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 corresponding to the same application source code, may require different amounts of resources.

The runtime switches that appear 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.
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). A side-effect of image flickering generated during system reconfiguration is acceptable. But when the application switches from a data driven scenario to another one, no side-effects that visibly affect the image of the channel being watched are acceptable.

As design methods for single and multi-task systems concentrate on different aspects, scenarios can also be classified in (i) *intra-task scenarios*, which appear within a sequential part of an application (i.e. a task); and (ii) *inter-task scenarios*, which represent operation modes of a multi-task application. 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 the methods presented above for each step of the methodology can be applied. For example, for hard real-time systems, scenario discovery can only use static analysis, and only detectors may be used to identify the current scenario at runtime, whereas for soft real-time systems predictors and statistical information from profilers may be used.

IV. Our trajectory for low energy design

This section presents our semi-automatic trajectory of discovery, predicting and exploiting application scenarios to reduce the energy consumed by a single task application on a dynamic voltage scaling (DVS) aware processor. The trajectory is adapted 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. The numerical results presented bellow, are obtained for an MP3 decoder running on a processor similar to an ARM7TDMI [14]

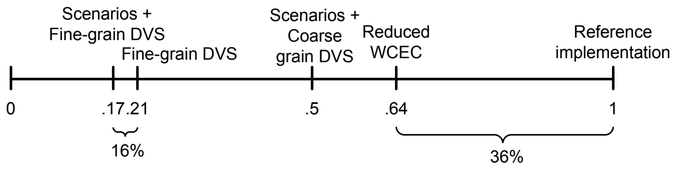


Fig. 3. Normalized energy for different MP3 hard real-time implementations

for which the frequency and the supply voltage can be set continuously within the operating range. A frequency change introduces a transition overhead of $70\mu s$ during which the processor stops running. References to papers that detail the methodology and the results are included.

A. Hard Real-Time

Our trajectory may generate different energy saving implementations, from a purely static one to an implementation that uses a fine grain DVS-aware scheduler. A comparison of their energy reduction is shown in figure 3 and discussed below.

In [15], we describe a method for the automatic discovery of scenarios that incorporate correlations between different parts of an application. These correlations 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, using the detected scenarios, the estimated worst case number of execution cycles (WCEC) for the entire application, which is the maximum between the ones obtained for each scenario, was reduced with 16%. 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, reducing the estimated WCEC with 15.9%, a processor with a 15.9% lower frequency is good enough. This saves 36% in energy consumption.

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 DVS-aware coarse-grain scheduler and selects once per loop iteration the supply voltage level. In the MP3 case, the average energy reduction is up to 50% from the original energy, depending on the input stream.

Our trajectory can also introduce scenarios in a fine-grain scheduler which changes the processor frequency multiple times during an iteration of the loop of interest. In [6], we showed that the combination of scenarios with a state-of-the-art fine-grain DVS-aware scheduler for hard real-time systems reduces the average energy with 16% compared to using only the DVS-aware scheduler. Fine-grain DVS gives better results than coarse-grain DVS if the frequency switching time is small enough. For larger switching times, fine-grain DVS is infeasible or coarse-grain DVS outperforms it.

B. Soft Real-Time

As for soft real-time systems a certain deadline miss ratio is acceptable, information collected by profiling the application can be used for scenario discovery and prediction. In [16], we describe a method and a tool that can automatically detect the most important application parameters and use them to define and dynamically predict scenarios. Using the generated code, the average over-estimation in the cycle budget required by the MP3 decoder to decode stereo songs is decreased with 46%, reducing the average reserved cycle budget for an audio sample from $3.9 \bullet 10^6$ to $3.5 \bullet 10^6$ cycles. The cost paid is 1.74% missed deadlines, but this can be reduced to 0% if an output buffer with the size of one audio sample is used. By using a proactive predictor that acts like a DVS-aware coarse-grain scheduler, the average energy reduction is 15% compared to the original soft real-time implementation. Up to 35% reduction is obtained if mono songs are also considered.

V. Conclusions

In this paper, we introduced the concept of *application scenarios*, which group operation modes that are similar from the resource usage perspective. Moreover, we presented their role in an embedded system design flow, illustrating the difference with the well known use-case scenarios.

Our scenario-based design trajectory for reducing the energy consumption under both hard and soft real-time constraints was presented. We showed that applying it on a benchmark, under different scheduling constraints, the energy consumption may be reduced with 16% to 50% compared to the state-of-the-art methods. To reduce energy, our trajectory exploits the difference in the 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 scenario discovery and clustering.

References

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

[2] M. B. Rosson and J. M. Carroll, "Scenario-based design," in *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. LEA, 2002, ch. 53, pp. 1032–1050.

[3] M. T. Ionita, "Scenario-based system architecting: A systematic approach to developing future-proof system architectures," Ph.D. dissertation, Technische Universiteit Eindhoven, Netherlands, May 2005.

[4] J. M. Paul, "Scenario-oriented design for single chip heterogeneous multiprocessors," in *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10*, 2005, p. 227b.

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

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

[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 et al., "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 et al., "Global memory optimisation for embedded systems allowed by code duplication," in *Proc. of SCOPES*, 2005.

[11] Peng Yang et al., *Multi-Processor Systems on Chip*. Morgan Kaufmann, 2003, ch. Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems.

[12] 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.

[13] S. Murali et al., "A methodology for mapping multiple use-cases onto networks on chips," in *Proc. of DATE*. IEEE, 2006.

[14] http://www.arm.com/products/CPUs/ARM7TDMI.html.

[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, "Profiling driven scenario detection and prediction for multimedia applications," in *Proc. of the IEEE Int. Conf. on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS)*, Greece, 2006, pp. 63–70.