Application Scenarios in Streaming-Oriented Embedded-System Design

S.V. Gheorghita, Twan Basten, Henk Corporaal

Institutions: Google, Eindhoven University of Technology

Published on: 01 Nov 2008 - IEEE Design & Test of Computers (IEEE Computer Society)

Topics: Design process, Frequency scaling, Energy consumption and Design methods

Application Scenarios in Streaming-Oriented Embedded-System Design

Stefan Valentin Gheorghita
Google Switzerland

Twan Basten and Henk Corporaal
Eindhoven University of Technology

A design method for handling increasingly dynamic real-time embedded-system applications can help developers cope with stringent system and market requirements. This method groups an application's operation modes into application scenarios and describes how to incorporate them in the overall design process. An automated scenario-based design trajectory reduces the energy consumption of a streaming application running on a single processor platform via dynamic voltage and frequency scaling.

■ **EMBEDDED SYSTEMS USUALLY** consist of processors that execute domain-specific applications. Much of their functionality is implemented in software, which runs on one or multiple processors, leaving only the high-performance functions implemented in hardware. Most typical embedded systems (such as TVs, cellular phones, and MP3 players) run multimedia or telecom applications that exhibit dynamic behavior, and their execution costs (such as the number of cycles and 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 could be a bit belonging to a compressed bitstream representing a coded video clip, or it could be a macroblock, video frame, audio sample, or network package. Usually, these applications must 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, such as a screen or speakers, and saves the application's internal state for further use; for example, in a video decoder, the previous decoded frame might be necessary to decode the current frame. The dynamism existing in modern applications leads to the use of different kernels for each stream object, depending on the object type. The actions executed in a particular loop iteration form the application's internal operation mode.

In this article, we describe a method that provides a systematic way of detecting and exploiting, at design time and runtime, the different internal operation modes. The fact that applications have different internal operation modes has not been fully exploited in embedded-system design thus far. Our approach combines a static analysis and profiling of the system at design time with information collected at runtime about the system's environment. By knowing a system's possible operation modes and information about their resource consumption at design time, it is possible to make specific and aggressive design decisions for each operation mode at different design steps.

To avoid complexity problems, we cluster the operation modes that are closely related to one another from a resource consumption perspective in application scenarios, distinguishing the truly different operation modes via different scenarios. It is then possible to derive a faster or lower-energy implementation (for example, by using different source-code optimizations per scenario) or a better estimation of required resources (such as the number of computation cycles or bandwidth). This leads to a smaller, less-expensive, and more energy-efficient system that can deliver the required performance.

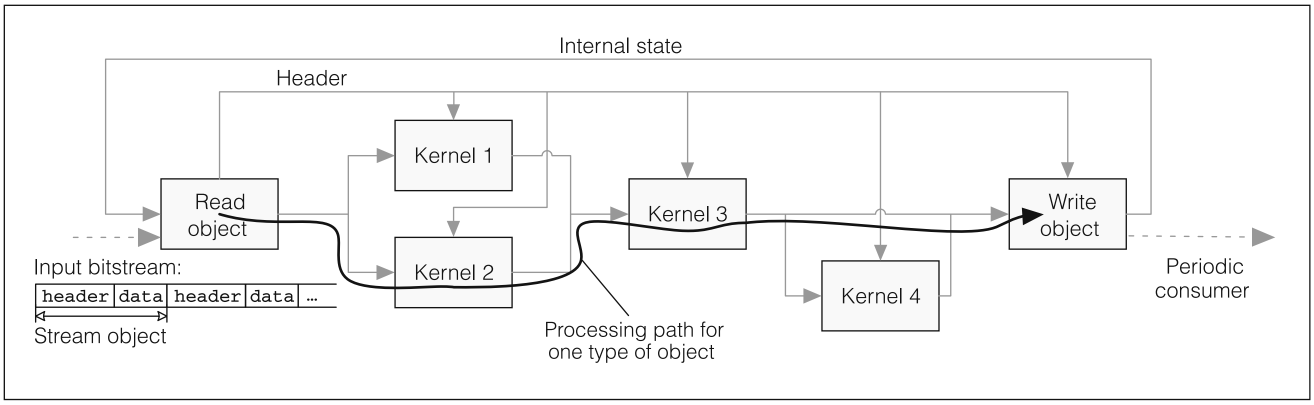


Figure 1. Typical streaming application processing a stream object.

Use-case versus application scenarios

Scenario-based design has been used for a long time in fields such as human-computer interaction and object-oriented software engineering.[1] In these cases, in an early phase of the development process, scenarios concretely describe a system's future uses. 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 perspective, all possible user actions and system reactions necessary to meet a proposed system functionality. These *use-case scenarios* characterize the system from the user perspective. In the embedded-systems area, they were used in both hardware and software design.

This article concentrates on *application scenarios*, which we derive from the system's behavior. These scenarios are a detectable set of an application's operation modes that are sufficiently similar in a multidimensional resource-based *cost space* (such as execution cycles, memory usage, or source code). They reduce system cost by exploiting information about what can happen at runtime to make better design decisions. Whereas use-case scenarios classify the application's behavior according to the various ways it can be used, application scenarios classify it from a resource-usage perspective. Researchers from IMEC were the first to identify and exploit these types of scenarios.[2]

The cost space is defined over a specific problem's dimensions of interest. 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. In order for us to exploit these modes, they must be detectable in the application, preferably as soon as the application starts to execute in one of them. Because the definition of an application scenario is very general, we must tailor it to the specific design problem at hand—for example, the application behavior for a specific type of input data.[3]

Figure 2 depicts a design trajectory employing use-case and application scenarios. It starts from a product idea, for which the stakeholders define the product's functionality as use-case scenarios. (The *stakeholders* are people, entities, or organizations that have a direct stake in the final system; they can be system owners, regulators, developers, users, or maintainers.) These scenarios characterize the system from a user perspective and serve as inputs to a design process that includes both software and hardware components. To optimize the system design, the detection and use of application scenarios augment this trajectory (bottom gray box in Figure 2). Once the application is coded, its scenarios related to resource utilization are semiautomatically extracted, and they are considered for the decisions made during the following phases of system design. The sets of use-case and application scenarios are not necessarily disjoint. One or more use-case scenarios could be merged in one application scenario, a use-case scenario might be split into several application scenarios, or several application scenarios might intersect several use-case scenarios.

As a case study, we consider the design of a portable MP3 player as a USB (Universal Serial Bus) stick. At first glance, there appear to be two main use-case scenarios: the player is connected to the computer, and music files are transferred between them; and the player is used to listen to music. But we can divide these scenarios into more detailed use-case scenarios, such as song selection, play, or fast-forward operations for the latter scenario.

Let us consider the play scenario. From a software perspective, we can split this use case into two different application scenarios: mono and stereo modes. By exploiting these scenarios, we can increase the system battery lifetime because mono mode requires less computing power. Thus, we can use a lower supply voltage while still meeting the decoding's timing constraints.

Application scenario methodology

The methodology for introducing application scenarios into the current embedded-system design trajectory has three steps (see Figure 3):

1. *Identification*. How is the application classified into scenarios?
2. *Predictor or detector derivation*. Which scenario does a particular iteration of the loop of interest belong to?
3. *Exploitation*. What can we do to optimize the system when it executes in a particular scenario?

These steps are influenced by the design context in which they are applied, but nevertheless are still general.

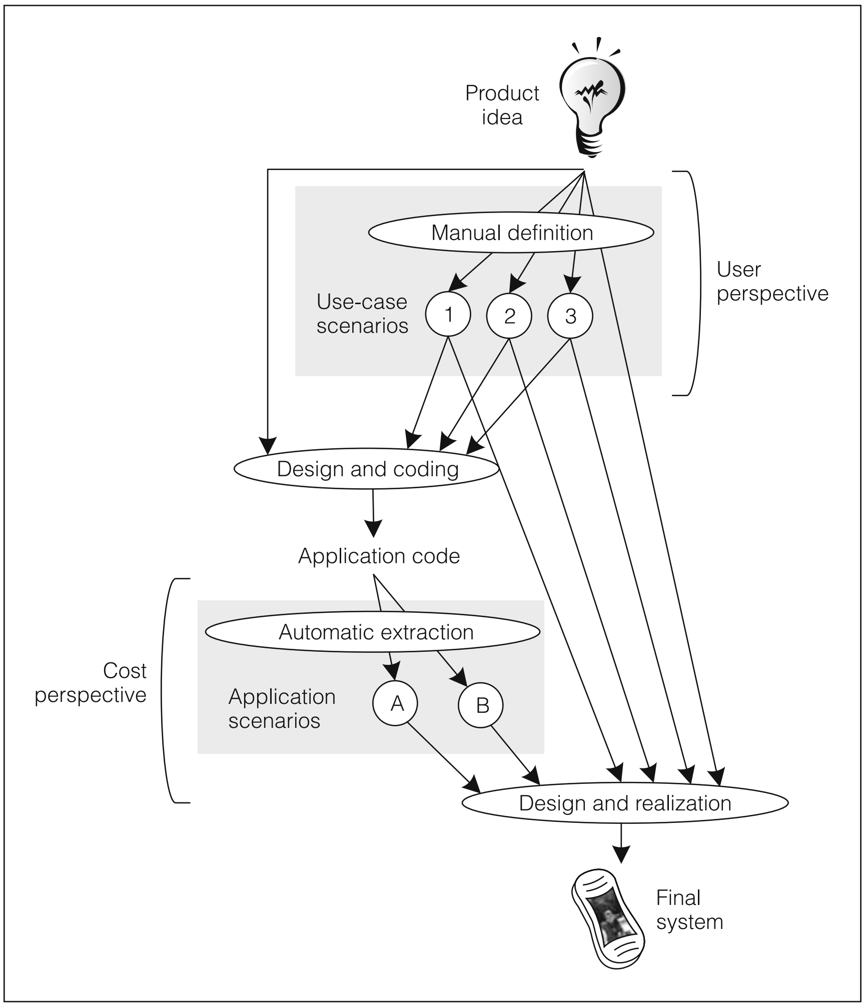
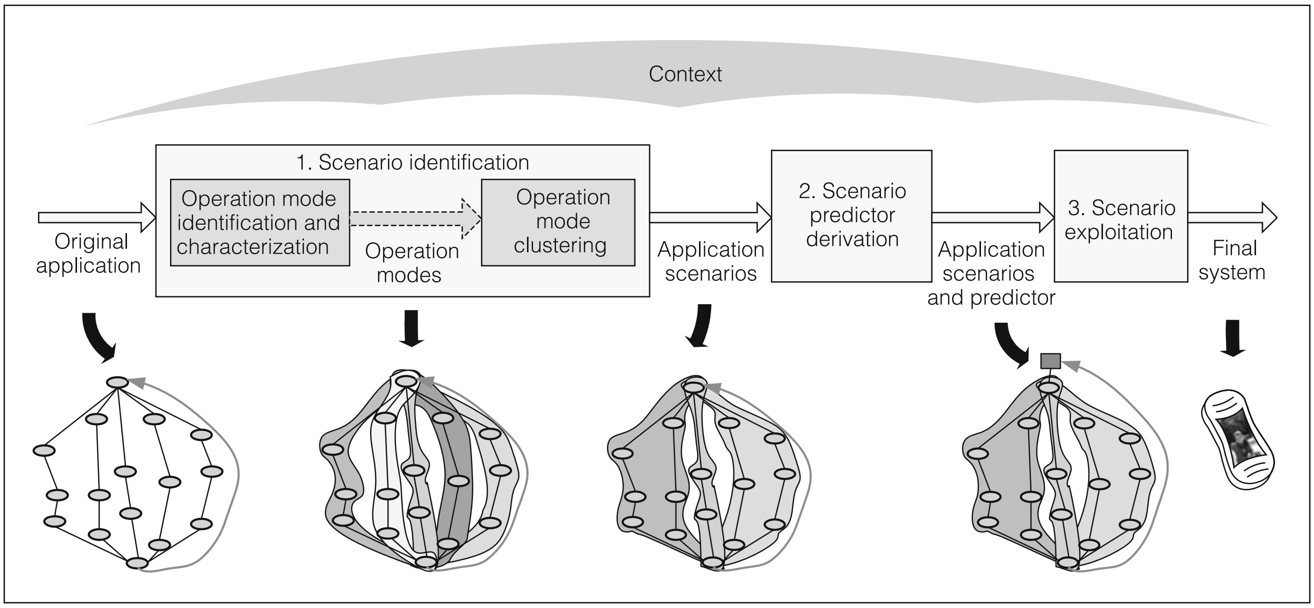


Figure 2. Full design flow.



Context
1. Scenario identification
Operation mode identification and characterization
Operation modes
Operation mode clustering
Original application
Application scenarios
2. Scenario predictor derivation
Application scenarios and predictor
3. Scenario exploitation
Final system

Figure 3. Scenario-based design flow for embedded systems.

Step 1

Scenario identification starts from the original application and identifies its different operation modes. These modes' resource usage (in the cost space of interest) is characterized, and those with similar needs are clustered in an application scenario. The methods used for scenario identification fall into three categories: *analytical*, *profiling*, and *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 because it cannot automatically collect information about how the application is really used and how it behaves at runtime—for example, which scenario is most frequently used at runtime. The application's real runtime behavior 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; in general, not all scenarios can be identified, because profiling might not cover all possible, distinct operation modes. To overcome this problem, we must consider a *backup scenario*, which 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 is the most powerful way to determine scenarios.

Especially for profiling and hybrid methods, an explosion in the number of operation modes could occur during their identification. In this case, it is necessary to make decisions 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 can 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, which first identifies a large set of the application's operation modes and afterwards tries to combine them based on similarity, there is also a top-down approach. This approach 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; they are just different ways of implementing scenario identification.

For the MP3 case study, we can look at a way to reduce a sequential streaming application's energy consumption by exploiting dynamic voltage and frequency scaling (DVFS) in modern processors. For hard real-time constraints, we use a static analysis for operation mode and scenario identification, augmented with a profiling approach for selecting the most common scenarios that appear during system use. For soft real-time constraints, profiling is the basis for identification because it can give the most accurate view of system utilization. However, for collecting internal application information (such as application structure or variables), we combine profiling with static analysis.

Step 2

Runtime scenario detector or predictor derivation is the step that determines which scenario the application runs in at a certain moment in time. We can either detect an application's current scenario on the basis of already known information (such as variable values) or predict it 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, the system switches from the current scenario to the new one, activating the optimizations for this new scenario.

There are different ways to implement predictors, such as static versus runtime adaptive or centralized versus distributed. Independent of the predictor implementation, we can use the following information: runtime application internal information such as variable values and executed code (for example, a basic block that appears only in one scenario); statistical information obtained through profiling or from the application designer (such as how often a scenario may appear at runtime); a probabilistic scenario transition model such as a Markov chain; and a history of active scenarios in the current execution. Predictors can 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 or extracted in the predictor source code.

The latter 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.

For the MP3 case study example, we can consider how to derive reactive centralized predictors using the application variables that do not change their values for one entire iteration of the loop of interest, under both soft and hard real-time constraints. The place where the predictor can be inserted into the application source code depends on the considered variables and on where assignments to these variables occur in the loop body. For hard real-time constraints, we extracted the predictor using static analysis, so it is in fact a static detector. For soft real-time constraints, we used the information collected via profiling to derive the predictor. To guarantee required system quality, we used a calibration mechanism that helps the predictor learn on the fly, making it a runtime-adaptive predictor.

Step 3

Scenario exploitation uses scenarios to optimize a design by applying more aggressive optimizations for each scenario. Because 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).

Yang et al. first explicitly identified the application scenario concept and used it to improve the mapping of dynamic applications onto a multiprocessor platform.[2] Other work applies this concept several times in an ad hoc manner, with an emphasis on the exploitation of scenarios, mostly ignoring (automatic) identification and prediction. Sasanka, Hughes, and Adve concentrated on saving energy for a sequential application.[4] The targeted architecture has a single processor with reconfigurable components (such as the 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 unclear how scenarios are predicted at runtime.

An extension considers two simultaneous resources for scenario characterization.[5] 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, researchers have also used scenarios in the context of the geometrical-loop-transformation framework to extend the scope of the geometrical model's applicability.[6] This 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 systematically detect operation modes on the basis of profiling, and then they cluster them in scenarios on the basis of a trade-off between the number of memory accesses and the code size increase.

In the context of multitask applications, Lee, Yoo, and Choi also investigated using application scenarios to reduce the energy consumed by a multitask application mapped on a voltage-scaling-aware processor.[7] Murali et al. characterize scenarios according to different communication requirements (such as bandwidth and latency) and traffic patterns.[8] They present a method to map application communication to a network-on-chip architecture, satisfying the design constraints of each individual scenario.

Researchers have also used scenarios to improve the operating system. Mamagkakis, Soudris, and Catthoor presented a way of optimizing dynamic memory allocation for the IPv4 layer in an IEEE 802.11b wireless network application.[9] Their method employs different allocation algorithms for different scenarios, identifying these scenarios on the basis of the possible network package sizes.

Most of the work we've mentioned thus far emphasizes scenario exploitation for a given application, not for a class of applications, and does not go into detail on identification and prediction, which are important topics in our work.

Application scenario classification

The different classes of embedded systems (hard versus soft real time) and the design problem that is optimized lead to multiple possible criteria that we can use for scenario classification.

Considering how scenario switches are driven at runtime, two main scenario categories can be considered: dataflow and event driven. Dataflow-driven scenarios characterize different executed actions in an application that are selected at runtime based on the input data characteristics (such as the type of streaming object). Usually, each scenario has its own implementation within the application source code. Event-driven scenarios are selected at runtime on the basis of events external to the application, such as user requests or system status changes (for example, the battery level). They typically characterize different quality levels for the same functionality (or *quality* *scenarios*), which can be implemented as different algorithms or quality parameters in the same algorithm. The two types of scenarios can form a hierarchy. For different quality levels, a dataflow-driven scenario could 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 dataflow-driven scenarios, side effects are unacceptable, whereas in event-driven scenarios, different potential side effects might be acceptable. For example, a switch between quality scenarios in a TV could appear as an image format change (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 dataflow-driven scenario to another one, no side effects that visibly affect the image are acceptable.

Because design methods for single and multitask systems concentrate on different aspects, scenarios can also be classified as *intratask scenarios*, which appear within a sequential part of an application (that is, a task), and *intertask scenarios* for multitask applications. This classification is also hierarchical. Usually, the scenario in which a multitask application is running is derived from the scenarios in which each application task is currently running. Dataflow-driven intratask and intertask scenarios are conceptually the same from resource-use and runtime-switching perspectives, but they have different impacts on the intratask and intertask parts of the design flow, and they are typically exploited in different ways.

Finally, scenario usage differs for soft and hard real-time systems. Not all the methods we've presented for each step of the methodology can always be applied. For example, for hard real-time systems, scenario identification can use only static analysis, and only detectors can be used to identify the current scenario at runtime. But for soft real-time systems, we can use predictors and statistical information from profilers.

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

Our trajectory and experimental results

We've developed an automatic scenario-based trajectory for a specific design problem: the identification, prediction, and exploitation of application scenarios to reduce the energy consumed by a single-task streaming application running on a DVFS-aware processor. DVFS is an effective energy-saving technique that involves varying a processor's frequency and voltage at runtime according to processing needs. Most of the algorithms integrated in our tools are general, and they can be used for different problems. Our trajectory works for both hard and soft real-time constraints. It starts from an application written in C, because C is the language that is most used to write embedded-systems software. The trajectory generates the final energy-aware implementation in C as well.

Experimental setup

Our numerical results refer to the energy consumption of an Intel XScale PXA255 processor, measured using the XTREM simulator.[10] We consider that the processor frequency (f_{CLK}) can be set discretely within the processor's operational range in 1-MHz steps. The supply voltage (V_{DD}) is adapted accordingly:

$$f_{\mathrm{CLK}} = k\frac{(V_{\mathrm{DD}} - V_{\mathrm{T}})^2}{V_{\mathrm{DD}}}$$

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

MP3 decoder

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

Hard real time. Our trajectory can generate different energy-saving implementations, from a purely static one to an implementation that uses a fine-grained DVFS-aware scheduler. Figure 4 shows a comparison.

Because hard real-time systems require a conservative design approach, we developed a method and tool that uses a static analysis of the application source code to identify and exploit scenarios for reducing the overestimation in the worst-case number of execution cycles (WCECs) compared to existing methods.[11] The scenarios incorporated correlations that differentiate between the source code parts that never execute together in the same iteration of the loop of interest and the ones that can. To avoid an explosion in the number of detected scenarios, our tool extracts the correlations using only information about the automatically detected application parameters with a large impact on the execution time.

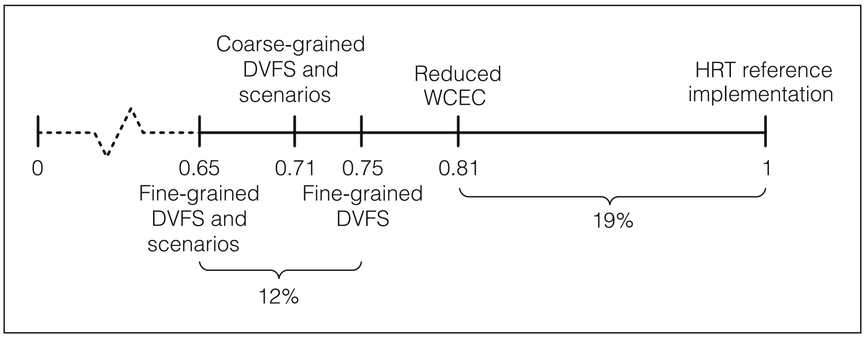


Figure 4. Normalized energy for different MP3 hard real-time implementations. (DVFS: dynamic voltage and frequency scaling; HRT: hard real time; WCEC: worst-case number of execution cycles.)

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

By exploiting the different WCECs for each scenario, we can further reduce the energy. Our trajectory can generate a proactive predictor that acts like a DVFS-aware coarse-grained scheduler. This scheduler selects the processor frequency and the supply voltage level once per loop iteration. For our benchmark, the energy reduction was 29% compared to the reference implementation.

Our trajectory can also augment scenarios in a fine-grained scheduler that changes the processor frequency multiple times during an iteration of the loop of interest. By combining scenarios with a state-of-the-art fine-grained DVFS-aware scheduler for hard real-time systems,[3] we reduced the average energy by 12% compared to using only the fine-grained DVFS-aware scheduler, and by 35% compared to the original reference.

Fine-grained DVFS gives better results than coarse-grained DVFS if the frequency-switching time is small enough relative to the period of the application loop of interest. For larger switching times, fine-grained DVFS is infeasible or coarse-grained DVFS outperforms it.

Soft real time. For soft real-time systems, a certain deadline miss ratio is acceptable, so using a conservative trajectory leads to an over-dimensioned system. Hence, we refined our trajectory to also consider information collected by profiling the application. In an earlier work,[12] we described a method and a tool flow that can automatically detect the most important application parameters to define and dynamically predict scenarios in soft real-time systems. The results of applying this trajectory to the MP3 case are summarized in Figure 5. The trajectory inserts three components into the application source code (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 because it is difficult, or almost impossible, during profiling to cover all possible operation modes in which a system might run. This mechanism 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 by 16% compared to the reference without scenarios, for a measured miss ratio of never more than one audio sample per 6 minutes (0.008%). Compared with the maximum energy reduction that can be obtained (see Oracle in Figure 5), we have reached about 50% of the

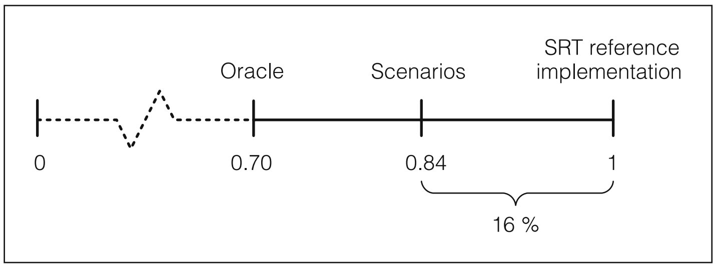


Figure 5. Normalized energy for different MP3 soft real-time implementations. (SRT: soft real time.)

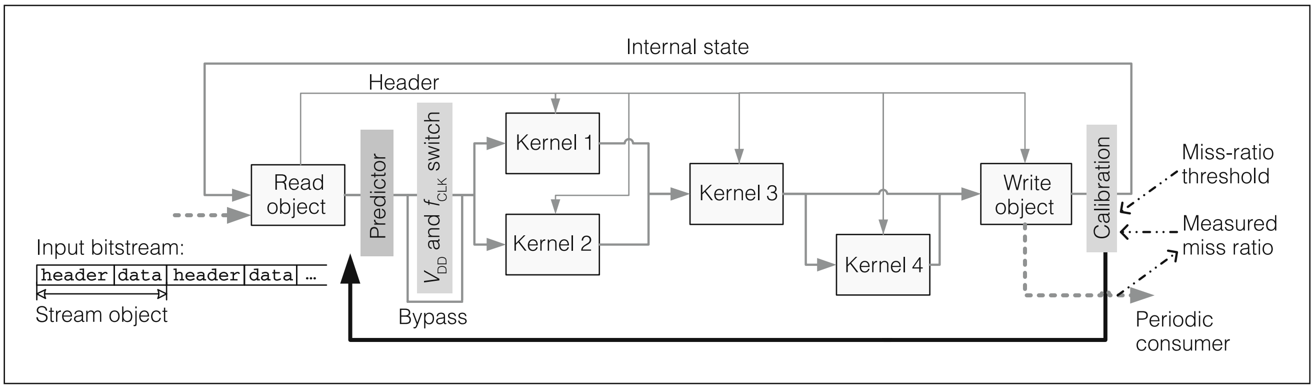


Figure 6. Final implementation for a soft real-time application. (Source: *J. Signal Processing Systems*, vol. 50, no. 2, Feb. 2008, pp. 137-161, "Scenario Selection and Prediction for DVS-Aware Scheduling of Multimedia Applications," S.V. Gheorghita, T. Basten, and H. Corporaal, Figure 15 © 2008 Springer.[12] With kind permission of Springer Science and Business Media.)

theoretically possible reduction. We plan to further reduce energy consumption by developing a more intelligent calibration mechanism.

Motion compensation video kernel

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

OUR TRAJECTORY EXPLOITS the difference in the required computation cycles between different scenarios, but it can be adapted to consider other resources, such as the number of memory accesses or memory size. It is interesting to further investigate the use of application scenarios in a multiprocessor context. Resource-usage estimation and scenario identification in a multiprocessor context is challenging, especially when considering multiple resources simultaneously. Also, the development of intelligent calibration mechanisms for application scenarios in various contexts with soft constraints is an interesting topic for future work. For more details about scenario-based design, visit http://www.es.ele.tue.nl/scenarios. ■

Acknowledgments

This work was supported by the Netherlands Organization for Scientific Research (NWO), project 612.064.101.

■ References

1. J.M. Carroll, ed., *Scenario-Based Design: Envisioning Work and Technology in System Development*, John Wiley & Sons, 1995.
2. P. Yang et al., "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems," *Proc. 15th Int'l Symp. System Synthesis* (ISSS 02), ACM Press, 2002, pp. 112-119.
3. S.V. Gheorghita, T. Basten, and H. Corporaal, "Intra-task Scenario-Aware Voltage Scheduling," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems* (CASES 05), ACM Press, 2005, pp. 177-184.
4. 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, Dec. 2002, pp. 144-155.
5. D.G. Sachs, S.V. Adve, and D.L. Jones, "Cross-Layer Adaptive Video Coding to Reduce Energy on General-Purpose Processors," *Proc. IEEE Int'l Conf. Image Processing* (ICIP 03), IEEE CS Press, 2003, pp. 109-112.
6. M. Palkovic, H. Corporaal, and F. Catthoor, "Global Memory Optimisation for Embedded Systems Allowed by Code Duplication," *Proc. Workshop Software and Compilers for Embedded Systems* (SCOPES 05), ACM Press, 2005, pp. 72-79.
7. S. Lee, S. Yoo, and K. Choi, "An Intra-task Dynamic Voltage Scaling Method for SoC Design with Hierarchical FSM and Synchronous Dataflow Model," *Proc. Int'l Symp. Low Power Electronics and Design* (ILSPED 02), IEEE Press, 2002, pp. 84-87.
8. S. Murali et al., "A Methodology for Mapping Multiple Use-Cases Onto Networks on Chips," *Proc. Design,*

Automation and Test in Europe Conf. (DATE 06), IEEE CS Press, 2006, pp. 1-6.
9. S. Mamagkakis, D. Soudris, and F. Catthoor, "Middleware Design Optimization of Wireless Protocols Based on the Exploitation of Dynamic Input Patterns," *Proc. Design, Automation and Test in Europe Conf.* (DATE 07), IEEE CS Press, 2007, pp. 118-123.
10. G. Contreras et al., "XTREM: A Power Simulator for the Intel XScale Core," *ACM Sigplan Notices*, vol. 39, no. 7, July 2004, pp. 115-125.
11. S.V. Gheorghita et al., "Automatic Scenario Detection for Improved WCET Estimation," *Proc. 42nd Design Automation Conf.* (DAC 05), ACM Press, 2005, pp. 101-104.
12. S.V. Gheorghita, T. Basten, and H. Corporaal, "Scenario Selection and Prediction for DVS-Aware Scheduling," *J. Signal Processing Systems*, vol. 50, no. 2, Feb. 2008, pp. 137-161.

Stefan Valentin Gheorghita is an engineer at Google Switzerland. He completed the work described in this article while pursuing his PhD in electrical engineering from the Eindhoven University of Technology, the Netherlands. His research interests include embedded systems, compilers, and parallel and distributed systems. He has a BSc and an MSc in computer science and engineering from Politehnica University of Bucharest, Romania, and a PhD in electrical engineering from the Eindhoven University of Technology. He is a member of the IEEE.

Twan Basten is an associate professor in the Electrical Engineering Department at the Eindhoven University of Technology, the Netherlands, and a Research Fellow of the Embedded Systems Institute, the Netherlands. His research interests include the design of resource-constrained embedded systems, with emphasis on multiprocessor systems and computational models. He has an MSc and a PhD in computing science from the Eindhoven University of Technology. He is a senior member of the IEEE and a life member of the ACM.

Henk Corporaal is a professor of embedded-systems architectures at the Eindhoven University of Technology. His research interests focus on the predictable design of soft and hard real-time embedded systems. He has an MSc in theoretical physics from the University of Groningen, and a PhD in electrical engineering from the Delft University of Technology.

■ Direct questions and comments about this article to Twan Basten, Eindhoven University of Technology, Dept. of Electrical Engineering, PO Box 513, 5600 MB Eindhoven, the Netherlands; a.a.basten@tue.nl.

For further information on this or any other computing topic, please visit our Digital Library at http://www.computer.org/csdl.