Journal of Signal Processing Systems 50, 137–161, 2008

DOI: 10.1007/s11265-007-0086-1

Scenario Selection and Prediction for DVS-Aware Scheduling of Multimedia Applications

S. V. GHEORGHITA, T. BASTEN AND H. CORPORAAL
EE Department, Electronic Systems Group, Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands

Received: 21 January 2007; Revised: 30 April 2007; Accepted: 23 May 2007

Abstract. Modern multimedia applications usually have real-time constraints and they are implemented using application-domain specific embedded processors. Dimensioning a system requires accurate estimations of resources needed by the applications. Overestimation leads to over-dimensioning. For a good resource estimation, all the cases in which an application can run must be considered. To avoid an explosion in the number of different cases, those that are similar with respect to required resources are combined into, so called *application scenarios*. This paper presents a methodology and a tool that can automatically detect the most important variables from an application and use them to select and dynamically predict scenarios, with respect to the necessary time budget, for soft real-time multimedia applications. The tool was tested for three multimedia applications. Using a proactive scenario-based dynamic voltage scheduler based on the scenarios and the runtime predictor generated by our tool, the energy consumption decreases with up to 19%, while guaranteeing a frame deadline miss ratio close to zero.

Keywords: dynamic voltage scheduling, soft real-time, application scenarios, embedded systems

1. Introduction

Embedded systems usually contain processors that execute domain-specific programs. Many of their functionalities are 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 and printers. The predominant workload on most of these systems is generated by stream processing applications, like video and audio decoders. Because many of these systems are real-time portable embedded systems, they have strong non-functional requirements regarding size, performance and power consumption. The requirements may be expressed as: the cheapest, smallest and most power efficient system that can deliver the required performance. During the design of these systems, accurate estimations of the resources needed by the application to run are required. Examples of resources include the number of execution cycles, memory-usage, and communication between application components.

Typical multimedia applications exhibit a high degree of data-dependent variability in their execution requirements. For example, the ratio of the worst case load versus the average load on a processor can be easily as high as a factor of 10 [27]. In order to save energy and still meet the real-time constraints of multimedia applications, many power-aware techniques based on dynamic voltage scaling (DVS) and dynamic power management (DPM) exploit this variability [17]. They scale the supply voltage and frequency of the processors at runtime to match the

changing workload. Taking into account that the processor energy consumption depends quadratically on the supply voltage ($E \propto V_{DD}^2$), whereas its execution speed (frequency) depends linearly on the supply voltage ($f_{CLK} \propto V_{DD}$), by reducing the processor speed to half, the energy consumption can be reduced to around a quarter.

Two main broad classes of voltage and frequency scaling techniques have been developed: (1) *reactive techniques*: after a part of the application is executed, the number of unused processor cycles[1] is detected and the processor frequency/voltage is reduced to take advantage of the unused computation power and (2) *proactive techniques*: detect or predict in advance that there will be unused cycles and set the processor frequency/voltage adequately. The proactive approaches are more efficient than the reactive ones, but they need a-priori derived knowledge about the input bitstream and/or the application behavior. This information can be included into the application itself as a *future case predictor* together with statically derived execution bounds for specific cases [10, 29], or it may be encoded like meta-data into the input bitstream during an offline analysis [2, 26]. To avoid an explosion in the number of different cases that are considered and in the amount of information inserted into the application or bitstream, not all different workloads are treated separately. Those that are similar with respect to required execution cycles are combined together into, so called, *application scenarios*.

Usually, to define scenarios for an application, its parameters (i.e., variables that appear in the source code) with the highest influence on the application workload are used. To the best of our knowledge, there is no way of automatically detecting these parameters, except for our previous work presented in [12, 13]. In this paper:

- We describe a method and a tool that can automatically identify the most important scenario parameters and use them to define and dynamically predict scenarios for a single-task soft real-time multimedia applications. When applied to three real-life benchmarks, the tool-flow identifies parameter sets that are similar to manually selected sets.
- We show how the method can be applied in a proactive DVS-aware scheduler, which, when applied to the three mentioned benchmarks, yields energy reductions up to 19%.

This method extends our previous work, overcoming the limitations of the static analysis for hard real-time systems used in [13], but it can not be applied to hard real-time applications, as the scenario detection and prediction is not always conservative. An earlier version of the current paper appeared as [12]. Compared to this, the current paper explains the automation of scenario selection, which was a manual step in [12], at the same time slightly generalizing the scenario concept. Moreover, to overcome the fact that our approach is not conservative, we describe a runtime mechanism that guarantees the application quality, as given by the percentage of deadline misses. Finally, we evaluate our tool-flow on a larger set of benchmarks and we quantify the amount of energy that was saved by using our approach in a proactive DVS-aware scheduler.

The paper is organized as follows. Section 2 surveys related work on scenarios and different power-aware approaches for saving energy for real-time systems, and presents how our current work is different. Section 3 presents how our approach fits in a general scenario based design methodology and the kind of multimedia applications that it can be applied to. Sections 4, 5, and 6 describe the three main steps of our approach of which an overview is given in Fig. 1. Section 7 presents the runtime calibration mechanism that is used for controlling the quality of the resulting application. In Section 8, our scenario detection and prediction method is evaluated on three realistic multimedia decoders. Conclusions and future research are discussed in Section 9.

2. Related Work

Scenarios have been in use for a long time in different design approaches [4], including both hardware [24] and software design [9] for embedded systems. 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 this work, we concentrate on a different kind of scenarios, so-called *application scenarios*, that

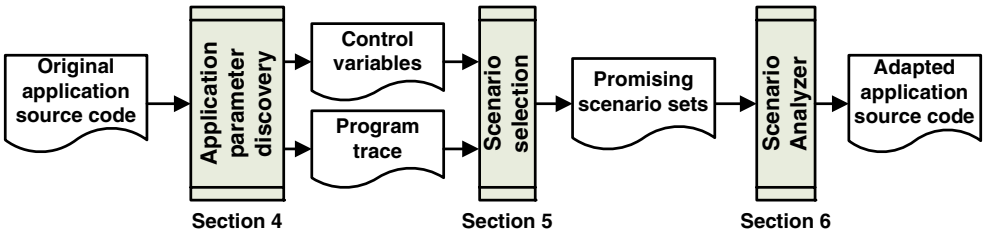

Figure 1. Tool-flow overview.

characterize the system from the *resource usage perspective*.

The application scenario concept was first used in [33] to capture the data-dependent behavior inside a thread, to better schedule a multi-threaded application on a heterogenous multi-processor architecture, allowing the change of voltage level for each individual processor. Other approaches that consider application scenarios to optimize a design include [22, 23, 28]. In [28], 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. To reduce the number of memory accesses, in [23], the authors selectively duplicate parts of application source code, enabling global loop transformations across data dependent conditions. They have a systematic way of detecting the most important application behaviors based on profiling and of clustering them into 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 [22], each scenario is characterized by different communication requirements (e.g., bandwidth, latency) and traffic patterns. The paper presents a method to map a multi-task application communication to a network on chip architecture, satisfying the design constraints of each individual scenario. Most of the mentioned papers (except [23]) emphasize on how the scenarios are exploited for obtaining a more optimized design and do not go into detail on how to select and predict scenarios. Our work focuses on these last two problems.

In the context of energy saving based on DVS/DPM techniques, two different approaches exist: reactive and proactive. The proactive approaches are more efficient than the reactive ones, as they can make decisions in advance based on the knowledge about the future behavior. In order to have this knowledge available at the right moment in time, several approaches propose to a-priori process the input bitstream of a multimedia application and add to it meta-information that estimates the amount of resources needed at runtime to decode each stream object (e.g., a frame). This information is used to reconfigure the system (e.g., using DVS) in order to reduce the energy consumption, while still meeting the deadlines. In [2, 15, 26] the authors propose a platform-dependent annotation of the bitstream, during the encoding or before uploading it from a PC to a mobile system. As it is too time expensive to use a cycle-accurate simulator to estimate the time budget necessary to decode each stream object, the presented approaches use a mathematical model to derive how many cycles are needed to decode each stream object. All these works aim at a specific application, with a specific implementation, and require that each frame header contains a few parameters that characterize the computation complexity. None of them presents a way of detecting these parameters, all assuming that the designer will provide them.

The other class of proactive approaches inserts into the application a workload case detector together with statically derived execution bounds for specific cases. The first approach for hard real-time systems was presented in [29]. It tries to predict in advance the future unused cycles, using the combined data and control flow information of the program. Its main disadvantage is the runtime overhead (which sometimes is big) that can not be controlled. In [10], we proposed a way to control this overhead, by using scenarios. We automatically detect the parameters with the highest influence on the worst case execution cycles (WCEC), and they are used to define scenarios. The static analysis used

in [10] is not very powerful, as it works for some specific cases only. It is also not really suitable for soft real-time systems, as the difference between the estimated WCEC and the real number of execution cycles may be quite substantial due to the unpredictability of hardware and WCEC analysis limitations. To overcome this issue, in [12], a profiling driven approach is used to detect and characterize scenarios. It solves the issue of manually detecting parameters in the soft real-time frame-based dynamic voltage scaling algorithms, like the one presented in [25]. In this paper, we extend the approach from [12] by making the tool-flow fully automatic and more robust, and by introducing into the resulting application a runtime mechanism that controls the application quality by keeping the number of deadline misses under a required bound. Moreover, instead of only quantifying the amount of cycle-budget over-estimation reduction, we look at energy saving for a larger set of benchmarks.

3. Overview of our Approach

This section starts by describing the characteristics of multimedia applications considered by our approach, and then details how our approach fits in the scenario methodology described in [11].

3.1. Multimedia Applications

Many multimedia applications are implemented as a main loop that reads, processes and writes out individual stream objects (see Fig. 2). A stream object might be a bit belonging to a compressed bitstream representing a coded video clip, a macroblock, a video frame, or an audio sample. For the sake of simplicity, and without loss of generality, from now on we use the word *frame* to refer to a stream object.

The read part of the application takes the frame from the input stream and separates it into a *header* and the frame's *data*. The process part consists of several kernels. For the processing of each frame, some of these kernels are used, depending on the frame type. The write part sends the processed data to the output devices, like a screen or speakers, and saves the internal state of the application for further usage (e.g., in a video decoder, the previous decoded frame may be necessary to decode the current frame). The dynamism existing in these applications leads to the usage of different kernels for each frame, depending on the frame type. The actions executed in a particular loop iteration form an internal *operation mode* of the application. Moreover, these applications have to deliver a given throughput (number of frames per second), which imposes a time constraint (deadline) for each loop iteration. In case of soft real-time applications, a given percentage of deadline misses is acceptable.

3.2. Scenario-aware Energy Reduction

The scenario methodology described in [11] consists of three main steps, presented in Fig. 3, each of them answering to a specific question:

1. *Identification*: given an application, how is it classified into scenarios?
2. *Prediction*: given an operation mode, to which scenario does it belong?
3. *Exploitation:* given a particular scenario, what can be done to optimize the application cost in term of resource usage?

Our approach follows this methodology, in the context of saving energy using a coarse grain frame-based DVS-aware scheduling technique for soft real-time applications. In the first part of the *identification* step (*Operation mode identification and characterization*, Section 4) the common operation modes are identified and profiled. As we are interested in

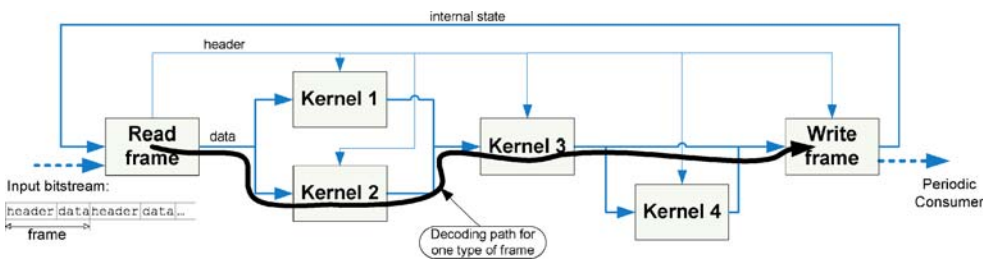

Figure 2. Typical multimedia application decoding a frame.

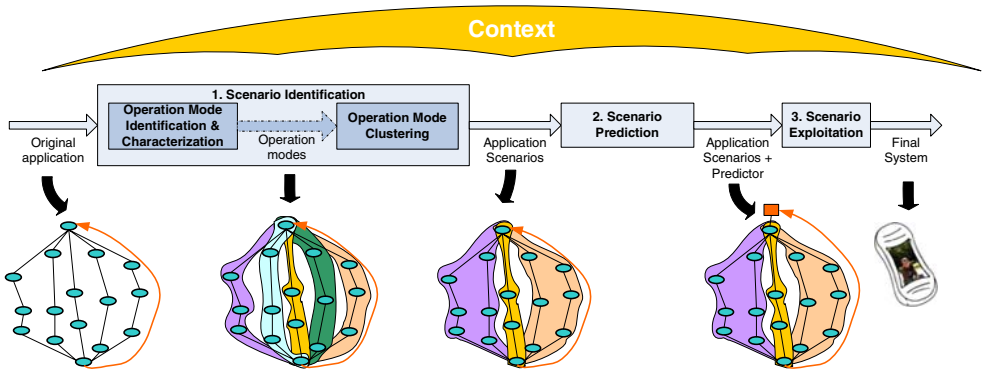

Figure 3. Application scenario usage methodology [11].

reducing energy by exploiting the different amounts of required computation cycles of different operation modes, we identify the application variables of which the values influence the application execution time the most, and we use them to characterize the operation modes. As the number of the operation modes depends exponentially on the number of control instructions in the application, the second part of the *identification* step (*operation mode clustering*, Section 5) aims to cluster the modes into application scenarios. The described clustering algorithm takes into account factors like the cost of runtime switching between scenarios, and the fact that the amount of computation cycles for the operation modes within a scenario should always be fairly similar.

In the *scenario prediction* step (Section 6) a proactive predictor is derived. Based on the parameters used to characterize the operation modes, it predicts at runtime in which scenario the application currently runs. As we aim to reduce the average energy consumption, in the *scenario exploitation* step, for each scenario, we compute the minimum processor frequency at which it can execute without missing the application's timing constraints. At runtime, when the predictor selects a new scenario, the processor frequency and supply voltage is adapted adequately. It leads to a coarse-grain schedule, as the processor frequency (and voltage) is changed once per scenario occurrence.

All the mentioned steps are based on profiling collected information, with the well-known limitation that the profiled information might not cover all operation modes that might occur. To overcome this limitation, a quality preservation mechanism is added to the final implementation of the application (Section 7). Its role is to keep the number of deadline misses under a required threshold.

4. Application Parameter Discovery

This section describes the first step of our method. The method is visualized in Fig. 1. As explained in the previous section, it is a concrete instance of the first two steps of the methodology shown in Fig. 3, where the application parameter discovery step corresponds to the first part of step 1 in Fig. 3. This section first explains how application parameters could be used to estimate the necessary cycle budget. The remaining parts of the section detail how these parameters are discovered by our method.

4.1. *Cycle Budget Estimation*

During system design, accurate estimations of the resources needed by the application in order to meet the desired throughput are required. This paper focuses on the cycle budget needed to decode a frame in a specific period of time (p_{frame}) on a given single-processor platform. This budget depends on the frame itself and the internal state of the application. In relevant related work [2, 15, 26], it is typically assumed that the cycle budget $c(i)$ for frame i can be estimated using a linear function on data-dependent arguments with data-independent, possibly platform dependent, coefficients:

$$c(i) = C_o + \sum_{k=1}^{n} C_k \xi_k(i), \tag{1}$$

where the C_k are constant coefficients that usually depend on the processor type, and the $\xi_k(i)$ are n arguments that depend on the frame i from the input bitstream.[2] Using for each frame its own transformation function with all possible source-code variables

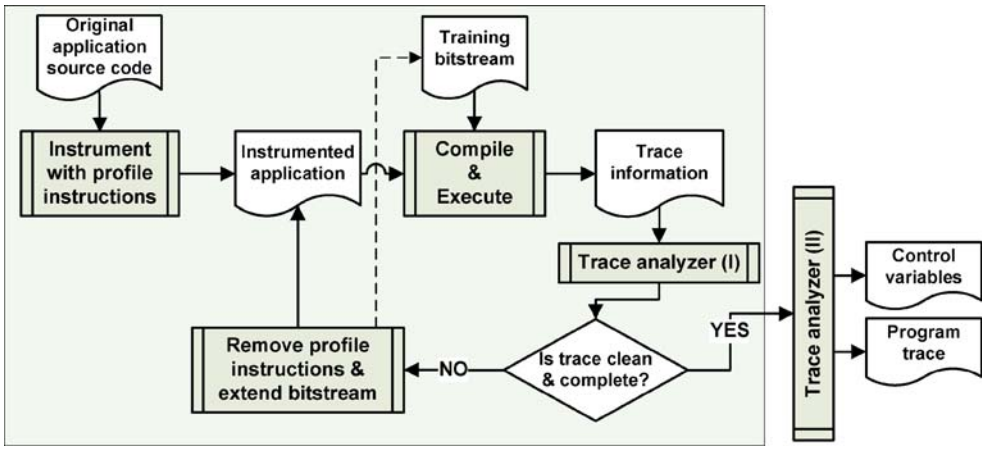

Figure 4. Tool-flow details for deriving application parameters.

as data-dependent arguments, gives the most accurate estimates. However, this approach leads to a huge number of very large functions. To reduce the explosion in the number of functions, the frames with small variation in decoding cycles are treated together, being combined in *application scenarios*. To reduce the size of each function, only the variables whose values have a large influence on the decoding time of a frame should be used. The following subsections present a method to identify these variables.

4.2. *Control Variable Identification*

The variables that appear in an application may be divided into *control variables* and *data variables*. Based on the control variable values, different paths of the application are executed, as they determine, for example, which conditional branch is taken or how many times a loop will iterate. The data variables represent the data processed by the application. Usually, the data variables appear as elements of large arrays, implicitly or explicitly declared. Attached to each array, there can be a control variable that represents the array size. Considering that each element of a data array is one data variable, it can be easily observed that, usually, there are a lot more data variables than control variables in a multimedia application.

The control variables are the ones that influence the execution time of the program the most, as they decide how often each part of the program is executed. Therefore, as our scope is to identify a small set of variables that can be used to estimate the amount of cycles required to process a frame, we separate the variables into data and control, based on application profiling. Moreover, we identify a subset of the control variables that do not influence the execution time and hence are not of interest to us. Both aspects are handled by the trace analyzer discussed in the next subsection.

The large gray box in Fig. 4 shows the work-flow for control variable identification. It starts from the application source code which is then instrumented with profile instructions for all read and write operations on the variables. The instrumented code is compiled and executed on a training bitstream and the resulting program trace is collected and analyzed. To find a *representative* training bitstream that covers most of the behaviors which may appear during the application life-time, particularly including the most frequent ones, is in general a difficult problem. However, an approach similar to the one presented in [19], where the authors show a technique for classifying different multimedia streams, could be used. The analysis performed on the collected trace information aims to discover if the trace contains data variables. If any are discovered, the profile instructions that generate this information are removed from the source code, and the process of compiling, executing and analyzing is repeated until the trace does not contain data variables anymore. As our method generates a huge trace if it is applied from the beginning on a large bitstream, we start with a few frames of the bitstream in the first iteration. At each iteration, we increase the number of considered frames as the size of trace information generated per frame reduces. The process is complete if the entire

training bitstream is processed and the resulting trace does not contain any data variables.

4.3. Trace Analyzer

The trace analyzer has two roles: (1) at each iteration of the flow for control variable identification, it identifies data variables and control variables that do not affect execution time substantially; and (2) when the process is complete, it generates the data necessary for the scenario selection step explained in Section 5 and a list of the remaining control variables.

The data variables that are declared as explicit arrays can be found via a straightforward static analysis of the source code. For the rest of the data variables, stored in implicitly declared arrays (e.g., the variable *a* from the source code of Fig. 5), the trace analyzer applies the following rule: if in the trace information generated for each frame, there is a program instruction that reads or writes a number of different memory addresses (e.g., the instructions from lines 3 and 4 in Fig. 5) larger than a threshold, we consider that all these memory addresses are linked to data variables, as this operation looks like accessing a data array. For this decision, we do not look for a specific array access pattern (e.g., a sequential access pattern as in line 3 or a random access pattern as in line 4 of our example). The profiling in combination with a threshold allows to differentiate between implicitly declared arrays that store data or control variables. This can not be obtained only by inspecting the source code, due to the complexity of the C language and the limitation of existing static analysis techniques, like pointer alias analysis [14]. Based on practical experience, we observed that the threshold is quite low. It is a configuration parameter for our tool, and its default value is four, as it is the appropriate value found by us in practice.

Loop iterators are the control variables that we consider to have only a small influence on the application execution time and that are easy to identify based on the trace information generated for each frame. These variables are not used to decide how many times a loop iterates; they just count the number of iterations. For example, in the piece of code of Fig. 5, the variable *n* bounds the number of iterations, while the loop iterator *i* counts them. Variable *n* might be of interest, but *i* is not. If there is a program instruction that writes the same variable more than once, this variable can be considered a loop iterator.[3]

```
     void process(char *a, int n) {
1      int i = 0;
2      while(i<n) {
3        f(a[i]);
4        f(a[a[i]]);
5        i++;
6      }
7    }
```

Figure 5. An educational example.

When the trace analyzer finishes, all data variables and loop iterators are removed. The trace analyzer generates a list with the remaining variables from the trace which are candidates for the ξ_k used in Eq. (1). During the scenario analyzer step (Section 6), their number is further reduced. Figure 6 shows the categories into which the application variables are divided, where category (b) covers the variables removed during the scenario analyzer step.

Besides the write and read operations, the program trace contains also the number of cycles needed to decode each frame (part of the operation mode characterization in step 1 of Fig. 3). This information is used in the scenario selection step, discussed in the next section.

5. Scenario Selection

This section presents our scenario selection approach (the second step in Fig. 1 and part 2 of step 1 in Fig. 3). It first details the scenario selection problem. It then continues in Section 5.2 by introducing the frame and scenario signatures that capture all the relevant information needed for scenario selection and prediction. The remaining part of the section describes the actual scenario selection step, which is detailed in the left gray box of Fig. 7. It consists of two main processes: (1) using a heuristic approach, multiple scenario sets are generated from the information previously derived by profiling the training bitstream (Section 5.3), and (2) from the generated scenario sets the most promising ones from an energy saving point of view are selected (Section 5.4).

5.1. The Scenario Selection Problem

In [12], scenarios are manually identified based on a graphically depicted distribution histogram that shows on the horizontal axis the number of cycles needed to decode a frame and on the vertical axis how often this cycle budget was needed for the training bitstream. Each identified scenario *j* is

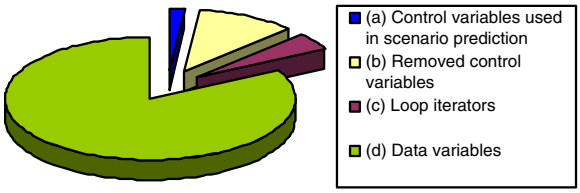

Figure 6. Variable distribution for MP3.

characterized by a cycle budget interval $(c_{lb}(j), c_{ub}(j)]$ that bounds the number of cycles needed to decode each frame that is part of the scenario. The set of identified scenarios covers all the frames that appear in the training bitstream.

In the final application source code generated by our method, for each frame of a scenario, c_{ub} is used as an estimate for the required cycle budget for processing it. So, each scenario introduces an over-estimation that is determined by the difference between c_{ub} and the average amount of cycles needed to process the frames belonging to it. As the aim is to exploit DVS, for each scenario the targeted processor frequency is set to the lowest frequency that can deliver at least c_{ub} cycles within a p_{frame} period of time. An overhead of maximum t_{switch} seconds[4] is taken into account for changing the processor frequency at runtime, when the application switches between scenarios. So, tight bounds c_{ub} and limited switching frequency are important.

Manual scenario selection is a time-consuming iterative job. The process starts by deriving an initial set of scenarios from the distribution histogram. Then, its quality in prediction and over-estimation is evaluated. It might not be straightforward to unambiguously characterize the manually selected scenarios by means of the variables identified in the previous section. Based on the obtained results, the set can be adapted and re-evaluated as often as necessary. A manual selection approach, similar to the one presented in [12], can easily exploit the information that can be extracted from the distribution histogram: (1) how often scenarios occur at runtime and (2) the introduced cycle-budget over-estimation. However, it is very difficult, even impossible, to take into account other necessary ingredients for selecting the best set of scenarios that are runtime detectable and introduce the lowest over-estimation, such as: (1) whether it is possible to distinguish at runtime between scenarios based on the considered control variables, (2) the possible overlap in the cycle budget intervals of identified scenarios, (3) how many switches appear between each two scenarios, and (4) the runtime scenario prediction and system reconfiguration (i.e., voltage/frequency scaling) overhead. All this information is taken into account in the heuristic algorithm presented in the following subsections. A running example, a simplified MPEG-2 motion compensation (MC) task, is used throughout the section for easier understanding.

5.2. *Scenario Signatures*

It is our aim to derive scenarios and scenario predictors from the knowledge that can be extracted from the training bitstream. To this end, we first characterize each frame from the training bitstream in terms of the control variables and its cycle count. This information is used in both the scenario selection and analyzer steps.

Let C be the set of control variables ξ_k obtained through the trace analyzer. *Frame signatures* are obtained by processing the trace generated for the training bitstream. For a frame i its signature $\Sigma_f(i)$ is defined as a pair:

$$\Sigma_f(i) = \big(V_{f(i)} = \{(\xi_k, \xi_k(i)) | \xi_k \in C\}, c(i)\big), \qquad (2)$$

where $\xi_k(i)$ is the value of control variable ξ_k for frame i, and $c(i)$ represents the number of cycles used to process frame i. For each frame, there can be some variables ξ_k that are not accessed during its processing, so they have undefined values. An example of a sequence of frame signatures for a training bitstream is shown in Fig. 7, where $\sim$ represents an undefined value.

Assume, for the moment, that all frames in the training bitstream have been partitioned into a set of scenarios. Let F_j be the set of all frames that belong to scenario j. A *scenario signature* can then be computed from the signature of all the frames in the training bitstream that are part of the scenario. Scenario signatures quantify the aspects of a scenario that are used in the scenario selection. For a scenario j, its scenario signature $\Sigma_s(j)$ is defined as a 4-tuple:

$$\Sigma_s(j) = ([c_{lb}(j), c_{ub}(j)], o(j), f(j), s(j)), \qquad (3)$$

where $c_{lb}(j) = \min_{i \in F_j}(c(i))$ and $c_{ub}(j) = \max_{i \in F_j}(c(i))$ bound the number of cycles needed to process each frame part of the scenario; $o(j) = \sum_{i \in F_j}(c_{ub}(j) -$

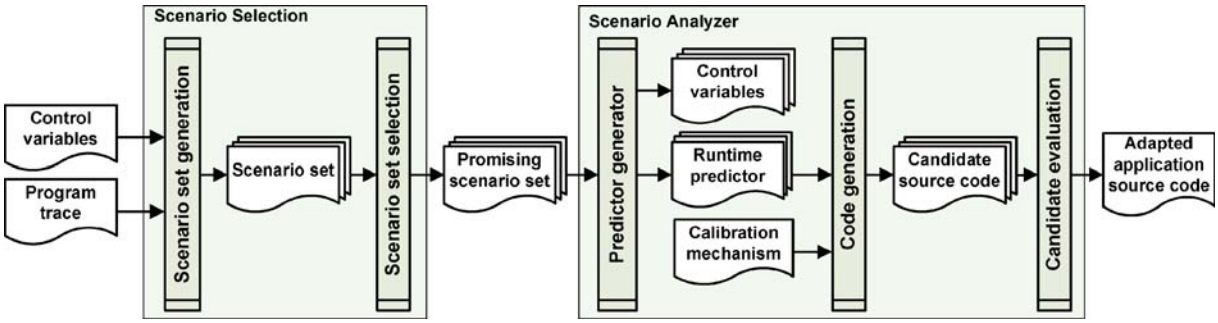

Figure 7. Tool-flow details for scenario selection and analyzer steps.

$c(i)$) represents the accumulated cycle budget over-estimation that this scenario introduces for the training bitstream; $f(j)$ counts how often the scenario appears (i.e., $f(j)$ equals the cardinality of F_j); and $s(j)$ counts how many times the application switches from this scenario to other scenarios (i.e., it counts in the training bitstream the number of frame intervals that consist of frames in scenario j). Figure 9a gives an example of two scenarios that contain some of the frames presented in Fig. 8.

The scenario selection algorithm repeatedly considers scenario candidates for clustering into one new scenario. To derive the signature for the scenario resulting from clustering a pair of scenarios (j_1, j_2), we introduce:

- $s(j_1, j_2)$ is the number of times that the application switches from scenario j_1 to scenario j_2 while processing the training bitstream, with $s(j_1, j_2) = 0$ if $j_1 = j_2$;
- $o(j_1, j_2)$ is the over-estimation introduced by clustering the two scenarios into a single one, where

$$o(j_1,j_2) = o(j_1) + o(j_2) + \begin{cases} (c_{ub}(j_1) - c_{ub}(j_2)) \cdot f(j_2), \text{ if } c_{ub}(j_1) > c_{ub}(j_2) \\ (c_{ub}(j_2) - c_{ub}(j_1)) \cdot f(j_1), \text{ if } c_{ub}(j_1) \leq c_{ub}(j_2) \end{cases} \quad (4)$$

Figure 9b gives a numerical example of how these functions are computed for the scenarios from Fig. 9a and the frame sequence given in Fig. 8.

Given two scenarios j_1 and j_2, with signatures $\Sigma_s(j_1)$ and $\Sigma_s(j_2)$, their *clustering* is a scenario $cls(j_1, j_2)$ with the signature:

$$\begin{aligned} &\Sigma_s(cls(j_1,j_2)) = \\ &\quad ([\min(c_{lb}(j_1), c_{lb}(j_2)), \max(c_{ub}(j_1), c_{ub}(j_2))], \\ &\quad o(j_1,j_2), \\ &\quad f(j_1) + f(j_2), \\ &\quad s(j_1) + s(j_2) - s(j_1,j_2) - s(j_2,j_1)). \end{aligned} \quad (5)$$

Figure 9c displays the scenario resulting from clustering the scenarios in Fig. 9a.

5.3. Scenario Sets Generation

This step, of which pseudo-code is shown in Fig. 10, represents the first part of the *scenario selection algorithm*. Its role is to divide the execution cases of the application in a number of scenarios. It receives as parameter the vector of frame signatures for the training bitstream. The algorithm returns multiple scenario sets, each of them covering all the given frames and being a potentially promising solution that represents a trade-off between the number of scenarios and the introduced over-estimation. More scenarios lead to less over-estimation. However, more scenarios lead to more switches and a larger predictor, which may increase the cycle overhead and enlarge the application source code too much.

$\Sigma_f(1) = (V_{f(1)} = \{(\xi_1,1),(\xi_2,\sim),(\xi_3,2)\},40)$
$\Sigma_f(2) = (V_{f(2)} = \{(\xi_1,2),(\xi_2,352),(\xi_3,2)\},39)$
$\Sigma_f(3) = (V_{f(3)} = \{(\xi_1,1),(\xi_2,\sim),(\xi_3,12)\},110)$
$\Sigma_f(4) = (V_{f(4)} = \{(\xi_1,2),(\xi_2,352),(\xi_3,12)\},112)$
$\Sigma_f(5) = (V_{f(5)} = \{(\xi_1,2),(\xi_2,352),(\xi_3,4)\},42)$
$\Sigma_f(6) = (V_{f(6)} = \{(\xi_1,2),(\xi_2,704),(\xi_3,2)\},39)$
$\Sigma_f(7) = (V_{f(7)} = \{(\xi_1,2),(\xi_2,704),(\xi_3,12)\},108)$
$\Sigma_f(8) = (V_{f(8)} = \{(\xi_1,2),(\xi_2,704),(\xi_3,4)\},41)$

Figure 8. A sequence of frame signatures.

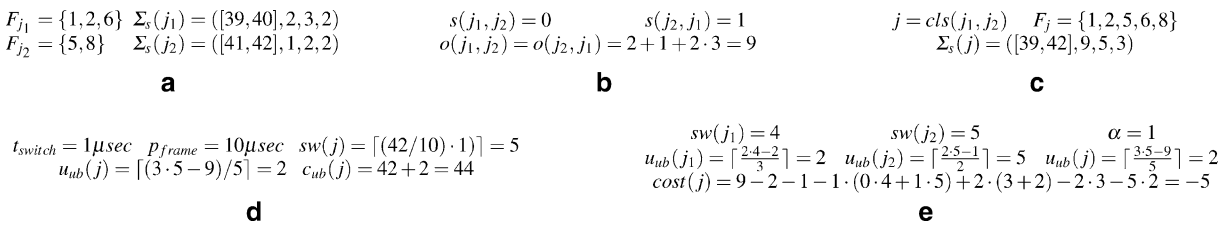

Figure 9. Example of scenarios. **a** Signatures, **b** functions, **c** clustering, **d** upper bound adaptation, and **e** clustering cost.

In the initialization phase (line 2), the algorithm generates an initial set of scenarios. It takes into account that there is no way to differentiate at runtime between two frames i_1 and i_2 if their signatures are such that $V_{f(i_1)} = V_{f(i_2)}$. So, in the initialization phase, all the frames i that have in the signature the same set $V_f(i)$ are clustered together in the same scenario.

The processing part of the algorithm starts with the initial set of scenarios and it is repeated until the scenario set contains only one scenario that clusters together all frames. At each iteration, the two most promising scenarios to be clustered are selected using a heuristic function, discussed in more detail below, and they are replaced in the scenario set by the scenario resulting from their clustering.

After the processing part, for each scenario j from each set of scenarios (lines 11–13), the upper bound of the cycle budget interval $c_{ub}(j)$ is adapted to accommodate, on average, the cycles spent to switch from this scenario to other scenarios. The maximum number of cycles used to switch from j is given by:

$$sw(j) = \lceil (c_{ub}(j)/p_{frame}) \cdot t_{switch} \rceil, \qquad (6)$$

where p_{frame} is the frame period, $c_{ub}(j)/p_{frame}$ is the processor frequency at which the scenario j is executed and t_{switch} is the maximum time overhead introduced by a frequency switching. In principle, the over-estimation introduced by a scenario can be used to accommodate for switching cycles. However, this over-estimation may be too small. Thus, if the over-estimation $o(j)$ introduced by the scenario is smaller than the total number of processor cycles needed to switch from it to other scenarios ($s(j) \cdot sw(j)$), then $c_{ub}(j)$ is incremented. Otherwise, it remains unchanged. The following formula computes the incrementing value:

$$u_{ub}(j) = \max\left(\left\lceil \frac{s(j) \cdot sw(j) - o(j)}{f(j)} \right\rceil, 0 \right). \qquad (7)$$

In Fig. 9d the cycle budget upper bound is recomputed for the scenario defined in Fig. 9c.

Recall that the aim of this work is to save energy. The tested heuristic functions for selecting which scenarios to cluster are based on cost functions that take into account: (1) the over-estimation of the resulting scenario, (2) the cycle budget upper bound adaptation that should be done for each scenario, and (3) the number of switches between scenarios and the switching overhead. Via the aspects (1) and (2), it is taken into account that the over-estimation introduced by a scenario could be used to compensate for the switching overhead from this scenario to other scenarios. There is a one-to-one correspondence between cost incurred by over-estimation cycles and cycles lost or gained via budget adaptation. Switching cost (aspect 3) will generally decrease when clustering scenarios. However, switching cost given in cycles should be weighted because the energy cost of these cycles depends on the ratio between the energy consumed during the frequency switching, information that can be taken from the processor datasheet, and the amount of energy used by normal processor operation during a period of time equal to t_{switch}. Considering all these aspects, the most promising clustering heuristic function that we found selects the pair of scenarios with the lowest cost taken as *over-estimation minus weighted switching plus adaptation*. Our experiments show that this cost function gives good results, while dropping any of

```
GENERATESCENARIOSETS(VECTOR frames)
1  solutions ← ∅
2  scenarioSet ←INITIALCLUSTERING(frames)
3  solutions.INSERT(scenarioSet)
4  while (scenarioSet.SIZE() ≠ 1)
5      do (j1, j2) ← GETTWOSCENARIOSTOCLUSTER(scenarioSet)
6         j ← CLUSTERSCENARIOS(j1, j2)
7         scenarioSet.REMOVE(j1)
8         scenarioSet.REMOVE(j2)
9         scenarioSet.INSERT(j)
10        solutions.INSERT(scenarioSet)
11 for each scenarioSet in solutions
12     do for each s in scenarioSet
13            do ADAPTSCENARIOBOUNDS(s)
14 return solutions
```

Figure 10. The scenario sets generation algorithm.

the three main aspects gives worse results. Formally, for scenarios j_1 and j_2 the clustering cost is given by:

$$\begin{aligned} cost(cls(j_1,j_2)) &= o(j_1,j_2) - o(j_1) - o(j_2) \\ &- \alpha \cdot (s(j_1,j_2) \cdot sw(j_1) + s(j_2,j_1) \cdot sw(j_2)) \\ &+ u_{ub}(cls(j_1,j_2)) \cdot (f(j_1) + f(j_2)) \\ &- u_{ub}(j_1) \cdot f(j_1) - u_{ub}(j_2) \cdot f(j_2), \end{aligned} \tag{8}$$

where α is a weighting coefficient for the number of cycles gained by reducing the number of switches. Figure 9e shows how the cost is computed for the two scenarios defined in Fig. 9a.

5.4. *Scenario Sets Selection*

This second and last step of the *scenario selection algorithm* aims to reduce the number of solutions that should be further evaluated, as the evaluation of each set of scenarios is a time-consuming operation. It chooses from the previously generated sets of scenarios the most promising ones. The goal is to find interesting trade-offs in cost (code size and runtime overhead) and gains (cycles and energy). Therefore, for making this decision, for each scenario set, the amount of introduced over-estimation and the number of runtime scenario switches are taken into account. Each solution is considered as a point in two two-dimensional trade-off spaces: (1) the number of scenarios (m) versus introduced over-estimation ($\sum_{j=1}^{m} o(j)$), and (2) the number of scenarios versus the number of runtime switches ($\sum_{j_1=1}^{m} \sum_{j_2=1}^{m} s(j_1,j_2)$). In the example given in Figs. 11 and 12 these points are called *generated solutions*. Each of the two charts is independently used to select a set containing promising solutions, and finally the two sets are merged. The selection algorithm consists of five steps:

1. For each chart, the sequence of solutions, sorted according to the number of scenarios, is approximated with a set of line segments, each of them linking two points of the set, such that the sum of the squared distances from each solution to the segment used to approximate it is minimized. This problem is an instance of the *change detection* problem from the data mining and statistics fields [5]. To avoid the trivial solution of having a different segment linking each pair of consecutive points, a penalty is added for each extra used segment. In Figs. 11 and 12, the selected segments and their end points are called approximation segments/points.
2. For each chart, we initially select all the approximation points to be part of the chart's set of promising solutions. These points are potentially interesting because they correspond to solutions in the trade-off spaces where the trends in the development in over-estimation (Fig. 11) and number of runtime switches (Fig. 12) change.
3. For each approximation segment from the over-estimation chart, its slope is computed. If it is very small compared to the slope of the entire sequence of solutions,[5] its right end point is removed from the set of promising solutions, as for similar over-estimation, we would like to have the smallest number of scenarios because that reduces code size and switches. In Fig. 11, for the segment between the solutions with four respectively six scenarios, the solution with six scenarios is discarded. The same rule does not apply for the *switches chart* because both end points are of interest. For a similar number of switches, the right end point represents the solution with the lowest over-estimation, and the left end point is the solution with the smallest predictor.
4. For each approximation segment from each chart, if its slope is larger than the slope of the entire sequence of solutions, intermediate points, if they exist, may be selected. They represent an interesting trade-off between the number of scenarios and the potential gains in over-estimation or number of switches. The percentage of selected points is chosen to depend on the ratio between the two slopes. In Fig. 12, the solutions with 28 and 29 scenarios are selected as intermediate points.
5. The sets of promising solutions generated for the trade-off spaces are merged, and the resulting union represents the set of the most promising solutions that will be further evaluated.

6. Scenario Analyzer

The scenario analyzer step is detailed in the right gray box from Fig. 7. It corresponds to the third step in Fig. 1, and it is an instance of step 2 of the general methodology of Fig. 3. It starts from the previous selected set of solutions, each solution being a set of scenarios that covers the whole application. For each solution, it generates: (1) for each scenario, an equation that characterizes the scenario depending

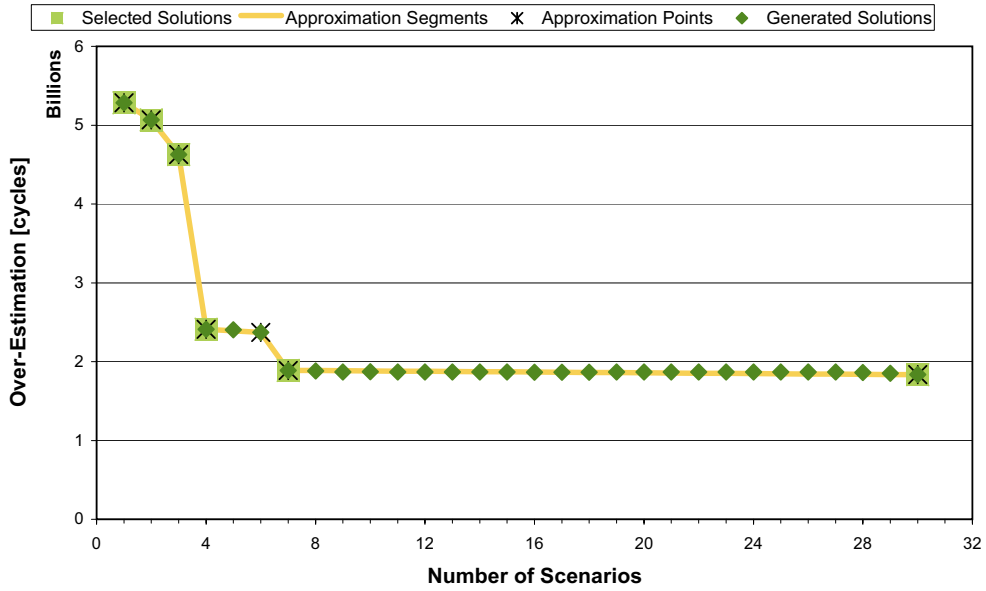

Figure 11. Scenario sets selection for MPEG-2 MC based on over-estimation.

on the application control variables; (2) the source code of the predictor that can be used to predict at runtime in which scenario the application is running; and (3) the list of the variables used by this predictor. The predictor together with the runtime quality calibration mechanism described in Section 7 is used to generate the source code for each solution. The best application implementation is selected by measuring the energy saving of each generated version of the source code on the training bitstream.

Scenario lcharacteristic function For each frame i, using its signature as defined in Section 5.2, a Boolean function $\chi_f(i)$ over variables ξ_k characterizing the frame is defined:

$$\chi_f(i)(\overrightarrow{\xi_k}) = \bigwedge_k (\xi_k = \xi_k(i)). \tag{9}$$

By using these functions, for each scenario j, a boolean function $\chi_s(j)$ over variables ξ_k characterizing the scenario is defined. Recall that F_j denotes the set of frames belonging to scenario j.

$$\chi_s(j)(\overrightarrow{\xi_k}) = \bigvee_{i \in F_j} \chi_f(i)(\overrightarrow{\xi_k}). \tag{10}$$

The canonical form of this Boolean function is obtained using the Quine McCluskey algorithm [20]. These functions can be used at runtime to check for each frame in which scenario the application should execute. Based on the *initial clustering* from the *scenario selection* step, at most one of these functions evaluates to *true* when applied to the control variable values of a frame. However, because these functions are computed based on a training bitstream, a special case may appear when a new frame i is checked against them: no scenario j for which $\chi_s(j)(\overrightarrow{\xi_k(i)})$ evaluates to *true* exists. In this case, the frame is classified to be in the so-called *backup scenario*, which is the scenario j with the largest $c_{ub}(j)$ among all the scenarios.

Runtime predictor The operations that change the values of the variables ξ_k are identified in the source code. Using a static analysis, for each of the possible paths within the main loop of the multimedia application, the instruction that is the last one to change the value of any variable ξ_k is identified. After this instruction, the values of all required variables are known. An identical runtime predictor is inserted after each of such instructions. This leads to multiple mutually exclusive predictors, from which precisely one is executed in each main loop iteration to predict the current scenario. An extension is to consider refinement predictors active at multiple points in the code to predict the current scenario: the first one will detect a set of possible scenarios, and

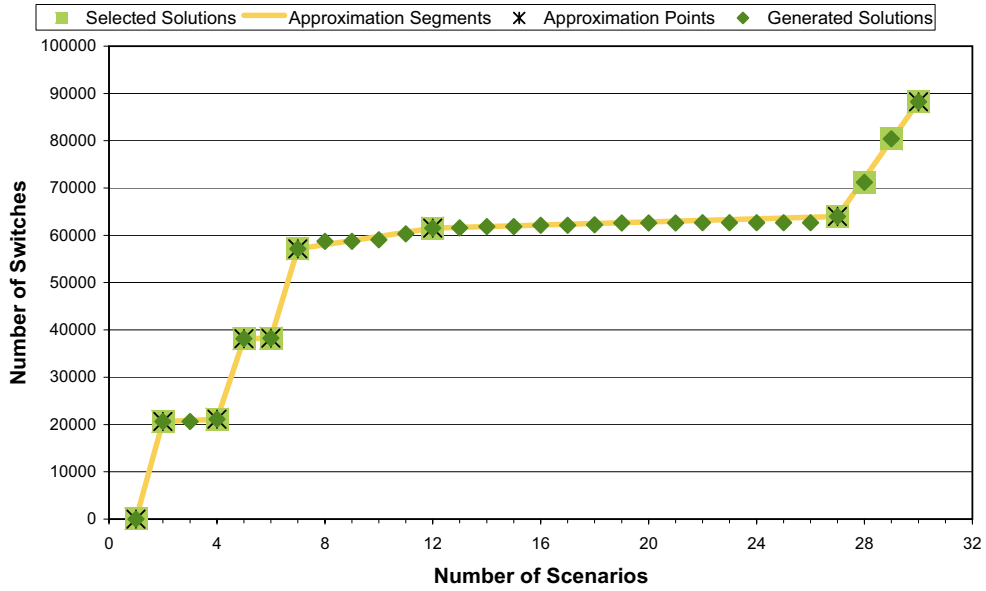

Figure 12. Scenario sets selection for MPEG-2 MC based on number of switches.

the following will refine the set until only one scenario remains. This extension might save more energy, as earlier switching between scenarios may be done. However, we leave this point open for future research.

We can use as the runtime predictor the scenario equations derived above. However, for a faster runtime evaluation, code optimization and the possibility of introducing more flexibility in the prediction, a decision diagram is more efficient. So, we derive the runtime predictor as a *multi-valued decision diagram* [32], defined by a function

$$f : \Omega_1 \times \Omega_2 \times ... \times \Omega_n \rightarrow \{1, .., m\}, \qquad (11)$$

where Ω_k is the set of all possible values of the type of variable ξ_k (including $\sim$ that represents undefined) and m is the number of scenarios in which the application was divided. The function f maps each frame i, based on the variable values $\xi_k(i)$ associated with it, to the scenario to which the frame belongs. The decision diagram consists of a directed acyclic graph $G = (V, E)$ and a labeling of the nodes and edges. The sink nodes get labels from $1, .., m$ and the inner (non-sink) nodes get labels from $\xi_1, ..., \xi_n$. Each inner node labeled with ξ_k has a number of outgoing edges equal to the number of the different values $\xi_k(i)$ that appear for variable ξ_k in all frames from the raining bitstream plus an edge labeled with *other* that leads directly to the backup scenario. This edge is introduced to handle the case when, for a frame i, there is no scenario j for which $\chi_s(j)(\overrightarrow{\xi_k(i)})$ evaluates to *true*. Only one inner node without incoming edges exists in V, which is the source node of the diagram, and from which the diagram evaluation always starts. On each path from the source node to a sink node each variable ξ_k occurs at most once. An example of a decision diagram for the sequence of frames of Fig. 8 is shown in Fig. 13a.

When the decision diagram is used in the source code to predict the future scenario, it introduces two additional cost factors: (1) *decision diagram code size* and (2) *average evaluation runtime cost*. Both can be measured in number of comparisons. To reduce the decision diagram size, a tradeoff with the decision quality is done. All the optimization steps done in our decision diagram generation algorithm (Fig. 14) are based on practical observations. The algorithm consists of five main steps:

1. *Initial decision diagram construction* (lines 1–21): For each scenario, a node is created and introduced in the decision diagram, and the node for the backup scenario is saved for future use (lines 2–4). For each node, the following information is stored: (1) the set of frames of the training bitstream for which the scenario prediction

process passes through the node, (2) its label (a control variable or a scenario identifier), (3) its type (SOURCE, SINK, and INNER) and (4) the variables that were not used as labels for the nodes on the path from the source node. For (SINK) nodes, the latter is irrelevant, and hence these nodes are assigned the empty set (line 3). A list with nodes that have to be processed is kept, and initially this list contains only the source node, unlabeled at this point (lines 5–6). While the list is not empty, the first node is extracted from it, and a variable that was not used on the path from the source to it is selected to label this node (lines 9–10). For each possible value for the selected variable that appears in the set of frames associated with the node (line 12), an edge is added in the decision diagram (line 19). In line 13, the set of frames for which the prediction process goes through node n and for which the value of ξ matches v is saved. The new edge is added either to a new inner node that will go in the list of nodes to be processed (lines 16–16), or to a scenario node, in which case the list of frames of the scenario node is updated (lines 17–18). The decision is made in line 14 by checking if the list of variables that were not used for deciding the path from the source to the current node contains only the variable selected for labeling the currently processed node. Finally, the node is inserted into the decision diagram and an edge from it to the backup scenario node is created (lines 20–21).

2. *Node merging* (line 22): Two inner nodes are merged if they have the same label and the set of the outgoing edges of one is included in the set of the other one. To understand the reason behind this decision, consider the decision diagram of Fig. 13a. It can be assumed that if $\xi_1 = 1$ and $\xi_3 = 4$ the application is, most probably, in scenario 2. This case did not appear for the training bitstream, but except for it the two ξ_3 labeled nodes imply the same decisions. If this assumption is made, the decision diagram can be reduced to the one shown in Fig. 13b.
3. *Node removal* (lines 23–24): The diagram is traversed and each node is checked to see if it really influences the decision made by the diagram. If it does not, it can be removed. An example of this kind of node can be found in Fig. 13b. In this diagram, it can be observed that whatever the values of ξ_1 and ξ_2 are, the current scenario is decided based on the value of ξ_3 (except for the values of ξ_1 and ξ_2 that did not occur in the training bitstream). This means that we can remove the nodes labeled with ξ_1 and ξ_2 from the diagram (see Fig. 13c). Note that if the values of ξ_1 and ξ_2 for a frame did not appear in the training bitstream, a scenario is selected based on the reduced diagram instead of the conservative backup scenario that would have been selected based on the original diagram.
4. *Interval edges* (lines 25–26): If a node has two or more outgoing edges associated to values

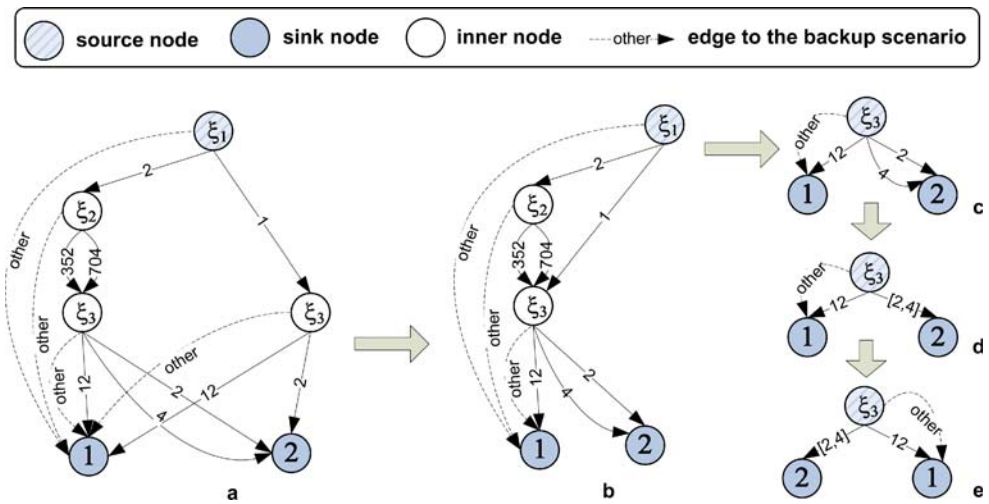

Figure 13. Simplified MPEG-2 MC decision diagrams: **a** original; **b** merging ξ_3; **c** removal of ξ_1 and ξ_2; **d** intervals; **e** reorder.

$v_1 < v_2 < .. < v_n$ that have the same destination, and there is no other outgoing edge associated with $v, v_1 < v < v_n$, then these edges may be merged in only one edge. In Fig. 13c, for both $\xi_3 = 2$ and $\xi_3 = 4$, scenario 2 is selected and there is no other value for $\xi_3 \in [2, 4]$ for which another scenario is selected. The assumption that if a value $\xi_3 \in [2, 4]$ appears for a frame, scenario 2 should be selected with high probability, leads to the diagram Fig. 13d.

5. *Edge reordering* (lines 27–28): To decrease the average runtime evaluation cost, the outgoing edges of each inner node are sorted in descending order based on the occurrence ratio of the values that label them. In Fig. 13e, the edges for the node labeled with ξ_3 were reordered, based on the observation that $\xi_3 \in [2, 4]$ appears most often.[6]

Different optimization steps of our tool may be disabled, so the tool may produce different decision diagrams, from the one created only based on the training bitstream (only steps 1 and 5 of the above algorithm) to the one on which all possible size reductions were applied (all five steps). Also, in each step of the algorithm, for example, the selection of variables for labeling nodes (line 9), different heuristics may be used. However, it might be possible that by applying all steps the prediction quality becomes bad. This may happen as the decisions made in our diagram generation algorithm are based on practical observations, and the application at hand might not conform to these observations. In this case, the steps that negatively affect the prediction quality should be identified and disabled.

For each predictor, the average amount of cycles needed at runtime to predict the scenarios is profiled on the training bitstream and the scenario bounds are updated to accommodate for this prediction cost. The process is similar to the one used in the previous section for accommodating for the scenario switching cost.

In the experiments presented in Section 8, we generated four fully optimized predictors, differentiated by:

– The variable selection heuristic for each node in step 1 of the algorithm (GETVAR, line 9 in Fig. 14): the variables with the most/least number of possible values are selected first. By selecting the one with most values first a lower runtime decision overhead might be introduced, as multiple small subtrees are created for each node and the decision height is reduced. On the other hand, by selecting the variable with the least possible values first, more freedom is given to the *interval edges* optimization step.
– The tree traversal in step 3 (TRAVERSENODE, line 23 in Fig. 14): breadth-/depth-first. Breadth-first tries to remove first the node, and then its children. Depth-first is doing the opposite.

All these four predictors can be used to achieve energy reduction, but there is no best one for all applications. Hence, in order to select the most efficient heuristics for an application, we generate the application source code for each of them. The structure of the generated source code is similar to the one presented in Fig. 15. It is derived from the original application, by introducing in it the predictor and the runtime quality preservation mechanism, which is described in the next section. Also, it contains the source code for adapting the processor frequency, which is activated only when the application switches from one scenario

```
NODE::NODE(SET frames, STRING label, NODETYPE type, SET vars);

GENERATEDECISIONDIAGRAM(SET frames, SET scenarios,
                        SCENARIO backup, SET vars)
 1  dd ← new DECISIONDIAGRAM()
 2  for each s in scenarios
 3      do dd.INSERT(new NODE(∅,s.name,SINK,∅))
 4  b ← dd.GETNODE(backup.name)
 5  nodes ← new LIST()
 6  nodes.PUSH(new NODE(frames,NIL,SOURCE,vars))
 7  while (nodes.SIZE() > 0)
 8      do n ← nodes.POP()
 9         ξ ← n.GETVAR()
10         n.label ← ξ.name
11         _vars ← n.vars − ξ
12         for each v in ξ.values
13             do _frames ← n.frames.GETFRAMES(ξ = v)
14                if (_vars ≠ ∅)
15                  then x ← new NODE(_frames,NIL,INNER,_vars)
16                       nodes.PUSH(x)
17                  else x ← dd.GETNODE(GETSCENARIO(_frames))
18                       x.frames ← x.frames ∪ _frames
19                n.ADDEDGE(v,x)
20         dd.INSERT(n)
21         n.ADDEDGE(OTHER,b)
22 dd.MERGESIMILARNODES()
23 for each n in dd.TRAVERSENODES()
24     do dd.TESTANDREMOVE(n)
25 for each n in dd.nodes
26     do n.REPLACEVALUEEDGESWITHINTERVALEDGE()
27 for each n in dd.nodes
28     do n.REORDEREDGES()
29 return dd
```

Figure 14. The decision diagram construction algorithm.

to another one. All the generated source codes are evaluated on the training bitstream and the one that gives the largest energy reduction is chosen. The variables used by its predictor are considered to be the most important control variables (Fig. 6).

7. Quality Preservation Mechanism

Because of the variation in the time spent in processing a frame, usually, in real-time embedded systems, an output buffer is implemented (see the right part of Fig. 15). The smallest possible buffer has a size equal to the maximum size of a produced output frame. The buffer is used to avoid the stalling of the process until the periodic consumer (e.g. a screen) takes the produced frame, allowing the start of the next frame processing before the current frame is consumed. To implement this parallelism, the conflict situation of producing a new frame before the previous one was consumed should be handled. This can be done (1) by using a semaphore mechanism that postpones the writing until the frame is consumed, or (2) by postponing the start moment of processing a new frame until it is sure that when the processing would be ready, the previous frame is already consumed.

We considered the second implementation, as there is no need for any synchronization mechanism. This gives more freedom in the consumer implementation and simplicity in output buffer implementation, for which a simple external memory may be used. Figure 16 explains how the start moment for frame processing is computed. For each frame i, S_i is defined as the earliest moment in time when the processing of frame i can start. It is equal to the moment when frame $i-1$ is consumed (D_{i-1}) minus the minimum possible processing time for each frame, estimated using static analysis as the best case execution time (BCET) or measured. The proactive DVS-aware scheduler that we used in our experiments makes sure that a frame i does not start earlier that S_i. The processing of frame i can however also not start until frame $i-1$ is ready (R_{i-1}). If the deadline of frame $i-1$ is missed, so $R_{i-1} > D_{i-1}$, depending on the application, one of the following two decisions can be made: (1) the processing of frame $i-1$ might be stopped at D_{i-1}, so the processing of frame i can start or (2) the application continues with the frame $i-1$ until it is ready, and then it starts with frame i. In the first case, which can for example be applied in an audio decoder, the processing of frame i actually starts at $\min(\max(S_i, R_{i-1}), D_{i-1})$. In the second one, typically used in video decoders that need a frame as a reference for the future, the processing of frame i starts at $\max(S_i, R_{i-1})$. For both ways to handle deadline misses, the consumer should not delete the frame from the output buffer when reading it, so it can read it again in case of a missed deadline. In our experiments from Section 8 we consider the first case, as it fits the best with the selected benchmarks.

As in our approach the cycle budget required by the application for a specific frame is predicted based on the information collected on a training bitstream, it is possible that the quality of the resulting system is lower than the required one, even when the above presented output buffer is exploited. This effect could appear because the training bitstream did not cover all the possible frames, so the scenario upper bounds might not be conservative. To keep the system quality under control, we introduce in the generated application source code a calibration mechanism (Fig. 15). This mechanism should be cheap in number of computation cycles and stored information size. We implemented it as in Fig. 17, and it adapts the table containing the scenario signatures used by the predictor. It counts the number of processed frames and the misses that appear in the system and for each scenario separately (lines 1–4). Also, for each scenario it stores (line 5) the maximum number of cycles that were used for processing a frame predicted to be in it. If the percentage of missed deadlines of the system is larger than a given threshold, the scenario with the largest number of misses is determined, and its cycle budget upper bound is updated (lines 6–11). As it was done also for the scenario switching mechanism and the predictor, the scenario bounds are updated to accommodate for the calibration mechanism too. Moreover, the overhead introduced by these three entities is taken into account when the cycle budget upper bound is updated at runtime (line 11).

8. Experimental Results

All the steps of the presented tool-flow were implemented on top of SUIF [1], and they are applicable to applications written in C, as C is the most used language to write embedded systems software. The resulting implementation for the appli-

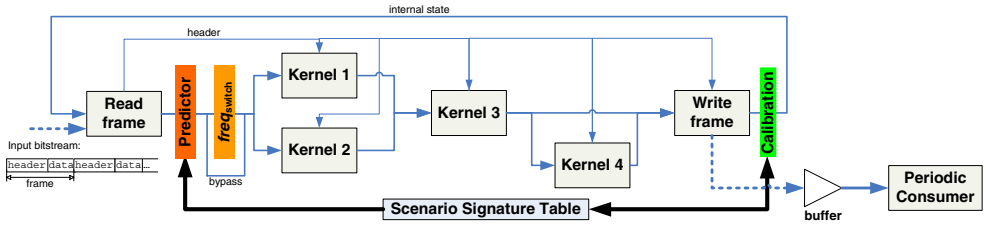

Figure 15. Final implementation of the application.

cation is written in C, and has a structure similar to the one presented in Fig. 15.

We tested our method on three multimedia applications, an MP3 decoder [18], the motion compensation task of an MPEG-2 decoder [21] and a G.72× voice decompression algorithm [30]. The energy consumption was measured on an Intel XScale PXA255 processor [16], using the XTREM simulator [7]. 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:

$$f_{CLK} = k \cdot \frac{(V_{DD} - V_T)^2}{V_{DD}}, \qquad (12)$$

where $V_T = 0.3V$ and the value of the constant k is computed for $V_{DD} = 1.5V$ and $f_{CLK} = 200MHz$. 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 equal with $4\mu J$ [3]. 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 start of a frame needs to be delayed, as explained in the previous section.

In the remaining part of this section, besides the main experiments that measure how much energy was saved by applying our approach, we quantify also the effect on energy of different steps of the decision diagram construction algorithm. Moreover, we investigate how the runtime calibration mechanism, different buffer sizes and different frequency/voltage switching costs influence the energy consumption and deadline miss ratio.

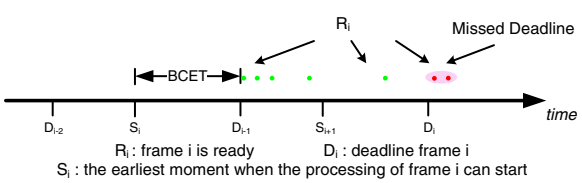

Figure 16. Output buffer impact on processing start time.

8.1. MP3 Decoder

The MPEG-I Layer III (MP3) decoder is a frame-based algorithm, which transforms a compressed bitstream in normal pulse code modulation data. A frame consists of 1,152 mono or stereo frequency-domain samples, divided into two granules. The standard specifies a fixed decoding throughput: a frame at each $26ms$. Details about the application structure and the source code are presented in [18]. To profile the application, we have chosen, as the training bitstream, a set of audio files consisting of: (1) the ones taken from [8], which were designed to cover all the extreme cases, and (2) a few randomly selected stereo and mono songs downloaded from the internet, in order to cover the most common cases. After removing the data variables and loop iterators, the number of remaining control variables ξ_k to be considered for scenario prediction is 41. This set of variables is far more complete than the one detected using the static analysis from [13]. The scenario sets generation algorithm of Section 5.3 leads to 2111 potential solutions (sets of scenarios). Using the method presented in Section 5.4, we reduced the size

```
CALIBRATESCENARIOTABLE(INT scen,INT cycles)
 1  framesCounter++
 2  if cycles > upperBound[scen]
 3     then appMissesCounter++
 4          missCounter[scen]++
 5          maxBudget[scen] ← max(maxBudget[scen],cycles)
 6          if appMissCounter/framesCounter > MISS-THRESHOLD
 7            then s ← scen
 8                 for i ← 1 to noScenarios
 9                     do if missCounter[s] < missCounter[i]
10                          then s ← i
11                 UPDATESCENARIOINTERVAL (s,maxBudget[s])
```

Figure 17. Runtime scenario quality control mechanism.

of the pool of solutions for which the predictor was generated to 34. This decreases the execution time of the scenario analysis (Section 6) from approximatively 4 days to less than 5 h. For each of the evaluated scenario sets, four fully optimized predictors were generated, as outlined in Section 6.

To quantify the energy saved by our approach, we measured the energy consumed by the resulting application via three experiments, by decoding (1) 20 randomly selected stereo songs, (2) 20 mono songs and (3) all these 40 songs together. These three categories are the most common combinations of songs that appear during an MP3 decoder usage.

The three groups of bars of Fig. 18 present the normalized results of our approach, evaluated for two miss ratio thresholds as used in the calibration mechanism: 1% and 0.1%. The energy improvement is given relatively to the energy measured for the case when no scenarios knowledge was used. In this case, the frame cycle budget is the maximum number of cycles measured for all input frames. In each decoding period, first the frame is processed, and then the processor goes in the idle state for the remaining time until the earliest possible start time for the next frame is reached.

We also compared our energy saving with the one given by an oracle (last bar of each group in Fig. 18), which is the smallest energy consumption that may be obtained. To compute it for a stream, all possible combinations of processor frequencies for decoding each frame from the stream were considered. The large difference between the energy reduction obtained by our approach and the oracle case is mostly due to the fact the oracle has a perfect knowledge of the remaining stream, based on which it may select different processor frequencies for the same scenario. Moreover, the oracle obtains an infinite accuracy without any cost, as it essentially considers any number of scenarios and variables for prediction, but has no prediction and calibration overhead. However, part of the energy difference is also due to the profiling drawbacks (e.g., not all possible samples were covered) and due to the lack of a better scenario bound adaptation mechanism (e.g., a mechanism that allows the reduction of a scenario cycle upper bound). These problems may be overcome by using a more efficient runtime calibration algorithm that may also decrease the scenarios bounds and even modify the decision diagram. This topic is left for future work.

An important evaluation criterion for our approach is the percentage of missed deadlines. As the energy savings may lead to a miss ratio that is too high, we use a runtime calibration mechanism that allows us to set a threshold for the miss ratio. To evaluate the effectiveness of the calibration mechanism and the overall approach, we measured the miss ratio in the experiments. Figure 19 shows the results for the two selected

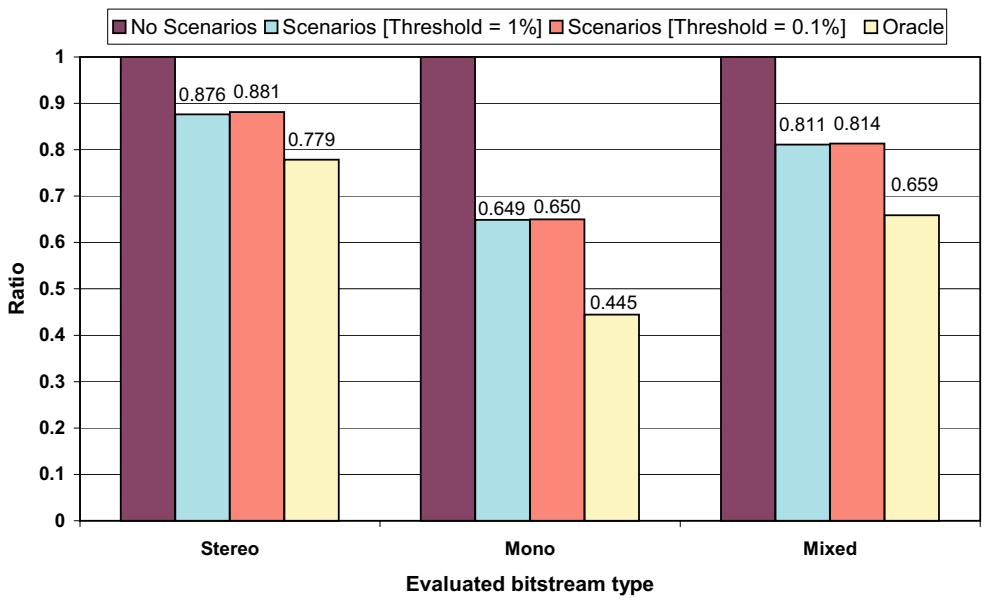

Figure 18. Normalized energy consumption for the MP3 decoder.

thresholds. There is a relatively large difference between the imposed threshold and the measured miss ratio. This is because the threshold is constrained before the output buffer, and the miss ratio is measured after it. The output buffer effect on miss ratio is hard to predict, but it will generally reduce the miss ratio. It can be observed that the combination of calibration and buffering is very effective.

Summarizing the main conclusions, for an MP3 player that is mainly used to listen mixed or stereo songs, the energy reduction that can be obtained by applying our approach is between 12% and 19%, for a miss ratio of up to one frame per 6 minutes (0.008%). The most energy efficient solution has 17 scenarios when decoding mixed (or only mono streams), and six when decoding only stereo streams.

Having concluded that our approach is effective, it is interesting to consider some of the design decisions in our approach, and some of the individual components in a bit more detail.

Recall that the decision diagram construction algorithm of Section 6 uses two heuristics, one for labeling nodes in the diagram and one for traversing the diagram during the reduction. This leads to four possible combinations. For all three experiments we did, the most efficient predictor was the one generated by selecting during the decision diagram construction first the variables with the least number of possible values and by using a breadth-first reduction approach. This combination is the most effective one in many cases, although in some of our later experiments also other combinations turn out to be the most effective one.

To show that the runtime calibration mechanism and all the steps that we used during the decision diagram construction are relevant for energy reduction, we did eight different experiments for a threshold of 0.1% using the set of mixed streams as the benchmark, as shown in Table 1. These experiments cover all possible cases for enabling/disabling three different components: (i1) the runtime calibration mechanism, (2) the node merging and removal (steps 2&3) in the decision diagram construction algorithm, and (3) the usage of interval edges in the algorithm (step 4). The node merging and removal were considered together because they are very tightly linked: by merging some nodes, other nodes become irrelevant as decision makers, so they can be removed.

The most important observation from Table 1 is that the merging and removal steps are essential to, and effective in, obtaining a substantial energy reduction. It turns out that when these optimization steps are omitted, 97% of the frames in the benchmark test falls into the backup scenario. This explains the low energy savings when the merging and removal steps are disabled. This also shows that the runtime prediction is not very effective in that case, which is in fact an indication that the training

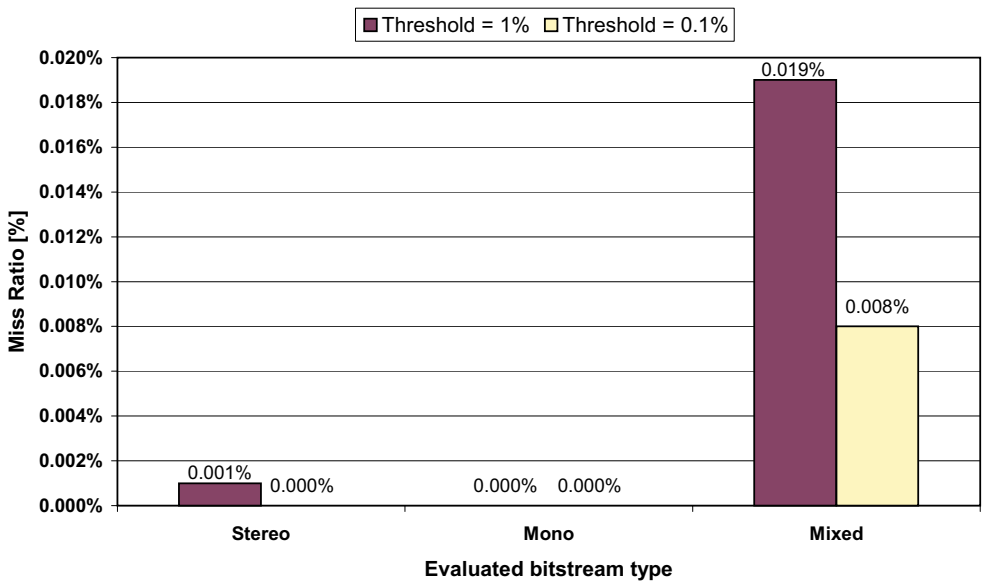

Figure 19. Miss ratio for the MP3 decoder.

bitstream was not sufficiently representative to obtain a good predictor (without these optimizations). An important conclusion from these experiments is that the optimization steps in the decision diagram construction algorithm provide a high degree of robustness to our approach. They effectively resolved the shortcomings of a poor training bitstream. The results furthermore show that also the interval optimization and the runtime calibration mechanism lead to further reductions in energy consumption. A final observation is that, for all the experiments, including the ones with the runtime calibration mechanism disabled, a set of scenarios and a predictor that meet the 0.1% miss ratio threshold was found. However, even if for this benchmark the required threshold could be met when the runtime calibration mechanism is not used, this will not be the case for all benchmarks and for all thresholds.

8.2. *MPEG-2 Motion Compensation*

An MPEG-2 video sequence is composed of frames, where each frame consists of a number of macroblocks (MBs). Decoding an MPEG-2 video can therefore be considered as decoding a sequence of MBs. This involves executing the following tasks for each MB: variable length decoding (VLD), inverse discrete cosine transformation (IDCT) and motion compensation (MC). Other tasks, like inverse quantization (IQ), involve a negligible amount of computation time, so we ignore them for the purpose of our analysis.

For our analysis, we use the source code from [21], and as a training bitstream we consider the first 20,000 MBs from each test file from [31]. As the IDCT execution time for each MB is almost constant, we focus on MC and VLD. In case of the VLD, our tool could not discover the parameters that influence the execution time, as they do not exist in the code. This task is really data dependent, reading and processing the input stream for each MB until a stop flag is met. For the MC task, the parameters found by our tool include all the parameters identified manually in [2], and which can be found in the source code. Observe that when knowledge characterizing frame execution times is introduced in frame headers, as for example, proposed in [26], our tool will be able to fully automatically detect the variables that store this information, and then exploit it to obtain energy reductions.

In the remainder of the experiment, we focus on the MC task, for which the processing period of a MB is $120\mu s$, which is very close to the frequency switching time $t_{switch} = 70\mu s$. Therefore, we analyzed the possibility of using different values for the weight coefficient α in the cost function of Eq. (8). A larger value will give higher importance to reducing the number of runtime switches, than to reducing the over-estimation, and it will usually result in smaller scenario sets. We evaluated all α values between one and six, and we found that the best energy saving may be obtained for $\alpha = 3$.

The evaluation of our approach in terms of energy on the full streams of [31] is shown in Fig. 20. Three miss ratio thresholds were evaluated, the two used for the previous experiment (1 and 0.1%), and an intermediate one (0.2%). For this application, the most energy efficient solutions use three scenarios for the 1 and

Table 1. Experimental results for MP3 with a threshold of 0.1% miss ratio.

Decision diagram construction		Runtime calibration	Selected predictor			Measured miss ratio (%)	Energy reduction (%)
Merging and Removal	Intervals		#Scenarios	Var. selection	Reduction		
X	**X**	**X**	**17**	**Least values**	**Breadth-first**	**0.008**	**18.65**
X	–	X	17	Least values	Breadth-first	0.008	15.46
–	X	X	67	Least values	–	0	1.08
–	–	X	67	Least values	–	0	1.08
X	X	–	17	Most values	Breadth-first	0.085	16.73
X	–	–	17	Least values	Breadth-first	0.008	15.46
–	X	–	67	Least values	-	0	1.11
–	–	–	67	Least values	–	0	1.08

0.2% miss ratio threshold, and two scenarios for the 0.1% threshold. The predictors were built by selecting, as for the MP3 decoder, first the variables with the least number of possible values, but using a depth-first instead of breadth-first reduction approach.

The measured miss ratio for all three thresholds is shown in Fig. 21. For a threshold of 0.2%, we obtained a 13% average energy reduction for all streams. The measured miss ratio was 0.09%, which represents one macroblock missed in every 13 frames when the video stream is in a QCIF format, that has a resolution of $176x144$ pixels.

If the threshold is pushed to 0.1%, the energy reduction drops to 3%, as for three of the 11 streams, it was very difficult to obtain this miss ratio. This is due to the considered buffer that can accommodate only a variation in execution of at most $18\mu s$, which is approximatively four times smaller than t_{switch}.

The results motivated us to do some experiments with varying buffer sizes and switching costs, to investigate their impact on energy savings and miss ratio. Table 2 shows the result of three experiments, the first one being the same experiment as reported in Figs. 20 and 21. It can be observed that a larger energy reduction for a 0.1% threshold (or any of the thresholds reported in Figs. 20 and 21) with a small measured miss ratio can be obtained when the frequency switching time t_{switch} is smaller or by increasing the output buffer size. The first might be obtained by using a different switching mechanism within the processor or another processor, and the second one is a viable solution when MC is considered in the context of a full MPEG-2 decoder. Then, the buffer size can be increased without a supplementary cost, as the decoder already has to store the entire frame.

As a final remark, it should be noted that, when MC is embedded in a complete MPEG-2 decoder, the relative energy reduction observed by our approach will decrease. Even though MC is the most energy hungry component in the decoder, it does not count for more than 50% of the total energy. However, as already mentioned, if knowledge about frame execution times is introduced in the headers, as in [2, 15, 26], our tool will be able to exploit this information to optimize more components of the decoder.

8.3. *G.72× Voice Decompression*

This benchmark [30] implements the decoders for a set of G.721/G.723 adaptive differential pulse-code modulation (ADPCM) telephony speech codec standards covering the transmission of voice at rates of 24, 32, and 40 kbit/s. Its input streams are sampled at the rate of 8,000 samples/s, so the deadline for each sample is $125\mu s$.

We analyzed our approach on the streams of [6], using as training bitstream 3,000 samples from each

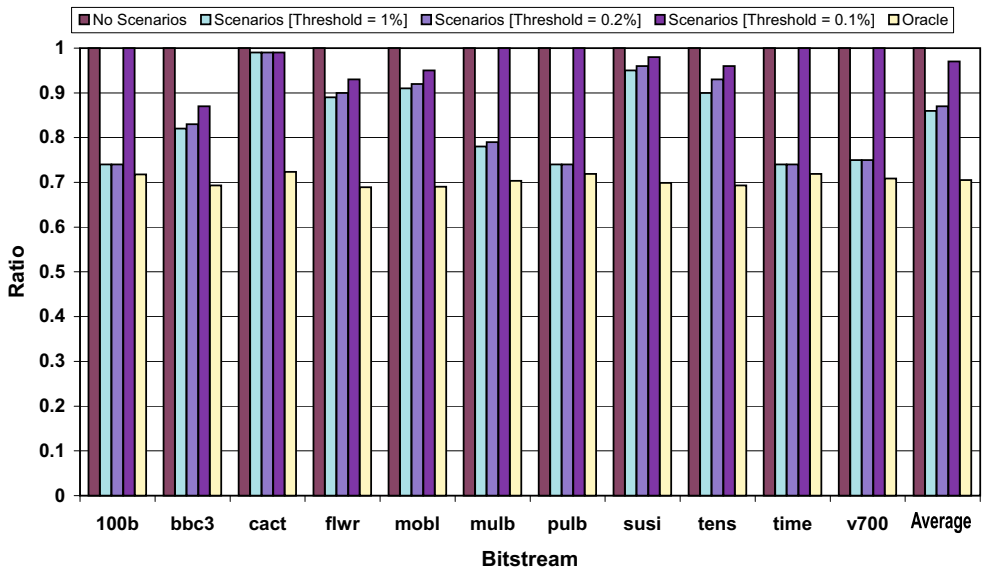

Figure 20. Normalized energy consumption for MPEG-2 MC.

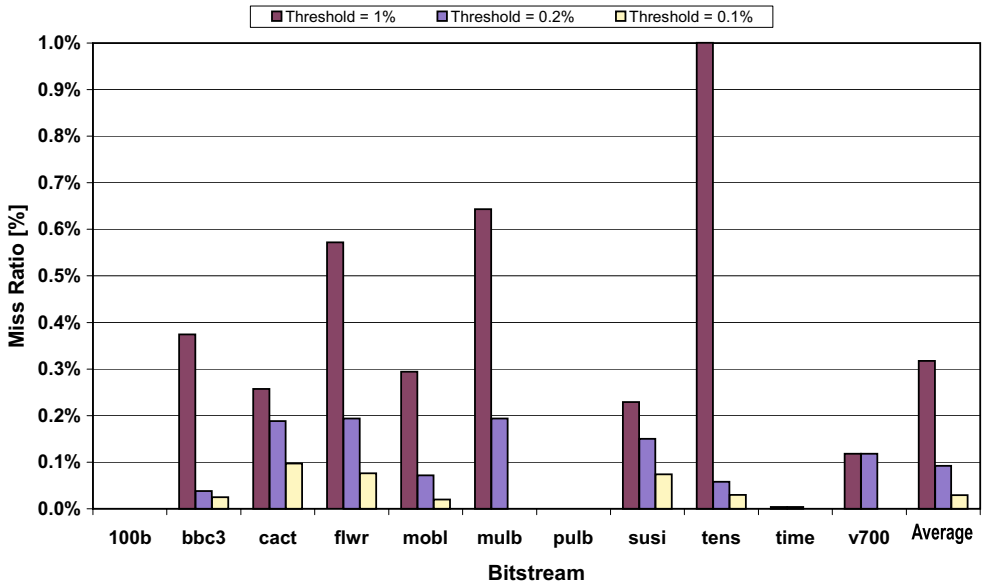

Figure 21. Miss ratio for the MPEG-2 MC.

test file. The best energy saving was obtained using a set of three scenarios, each of them associated with a specific voice transmission rate: 24, 32 and 40 kbits/s. Figure 22 shows the results, both detailed per input type, and averaged. As for each stream the transmission rate is fixed, the number of runtime switches is exactly one, namely the initial scenario selection for the first sample from the stream. This, together with the fact that only one parameter is used in scenario detection, which helped in having a fully representative training bitstream, leads to a miss ratio equal to zero for any imposed threshold. So, even if the resulting improvement is small (just 2%), it comes for free, without quality reduction. Furthermore, our method realizes close to 50% of the maximum theoretical possible improvement of slightly over 4%, computed via the oracle. The result of almost 50% of the theoretical maximum is inline with the earlier two experiments.

Table 2. Experimental results for MPEG-2 MC with a threshold of 0.1% miss ratio.

Buffer size (macroblocks)	t_{switch} (μs)	Energy reduction (%)	Measured miss ratio (%)
1	70	2.7	0.029
1	10	17.1	0
10	70	15.9	0.008

9. Conclusion

In this paper, we have presented a profiling driven approach to detect and characterize scenarios for single-task soft real-time multimedia applications. The scenarios are identified based on the automatically detected control variables whose values influence the application execution time the most. In addition, we present a technique to automatically derive and insert predictors in the application code, which are used at runtime to select the current scenario. Our method is fully automated and it was tested on three multimedia applications. For all of them, the identified sets of variables are similar to manually selected sets. We show that, using a proactive DVS-aware scheduler based on the scenar-

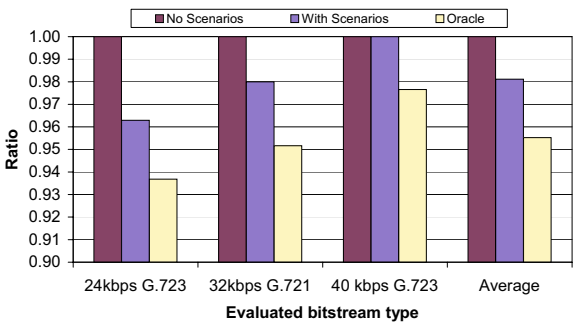

Figure 22. Normalized energy consumption for the G.72× voice decompression.

ios and the runtime predictor generated by our tool using the identified variables, energy consumption decreases with up to 19%, having guaranteed, using a simple runtime calibration mechanism, a frame deadline miss ratio of less than 0.1%. In practice, due to output buffering, the measured miss ratio decreases even to almost zero.

In future work, we would like to investigate different runtime calibration algorithms, that learn on the fly and adapt the scenario bounds, the number of scenarios and the decision diagram underlying the predictor. The information collected and processed by these control algorithms will be used not only for keeping the miss ratio under control, but also for further reduction in energy consumption. We also plan to extend our work to multi-task applications. Even if most of the basic steps of the presented trajectory (e.g., parameter identification, scenario prediction) remain unchanged, others, particularly scenario selection, have to be adapted to accommodate the specific problems that appear in multi-task applications (e.g., communication delay between tasks, pipelined execution). Moreover, scenario based design is not limited to multimedia applications and execution time estimation. It is interesting to investigate to what extent our techniques can be applied to other systems and/or other resource costs (such as memory accesses). Again, parameter identification and scenario prediction seem relatively straightforward to adapt. Scenario selection is the step that depends the most on the particular context.

Acknowledgment

This work was supported by the Dutch Science Foundation, NWO, project FAME, number 612.064.101. More information can be found on http://www.es.ele.tue.nl/scenarios.

Notes

1. The unused processor cycles represent the difference between how many cycles were estimated and how many were really needed by the application.
2. Eq. (1) could potentially have non-linear dependencies on the $\xi_k(i)$ (e.g., $\xi_k(i)^2$). For this paper, the function format is not relevant, as we only use the $\xi_k(i)$ to predict the program scenarios and not to estimate the cycle count.
3. The same behavior appears also in the case of counters, but we do not make the difference between counters and iterators, removing these variables in both cases.
4. t_{switch} can be extracted from the processor datasheet.
5. The sequence slope is the slope of the segment that links the first and the last point from the sequence.
6. Scenario 2 from the decision diagram is the same as the scenario j computed in Fig. 9.

References

1. S.P. Amarasinghe, J.M. Anderson, M.S. Lam, and A.W. Lim, "An Overview of a Compiler for Scalable Parallel Machines," in *Proc. of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, Germany, 1993, pp. 253–272.
2. A.C. Bavier, A.B. Montz and L.L. Peterson, "Predicting MPEG Execution Times," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, 1998, pp. 131–140.
3. T.D. Burd, T.A. Pering, A.J. Stratakos and R.W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System. *IEEE J Solid-State Circuits*, vol. 35, no. 11, 2000, pp. 1571–1580.
4. J.M. Carroll (Ed.) "*Scenario-based Design: Envisioning Work and Technology in System Development*". Wiley, New York, NY, 1995.
5. S.S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom "Change Detection in Hierarchically Structured Information," *ACM SIGMOD Record*, vol. 25, no. 2, 1996, pp. 493–504.
6. S.M. Clamen "8bit ULAW files collection," 2006. http://www.cs.cmu.edu/People/clamen/misc/tv/Animaniacs/sounds/
7. G. Contreras, M. Martonosi, J. Peng, R. Ju and G.Y. Lueh, "XTREM: A Power Simulator for the Intel XScale core. *ACM SIGPLAN Not.*, vol. 39, no. 7, 2004, pp. 115–125.
8. M. Dietz, et al., "MPEG-1 audio layer III test bitstream package," 1994. http://www.iis.fhg.de.
9. B. Douglass, "Real Time UML: Advances in the UML for Real-Time Systems," Addison Wesley, Reading, MA, 2004.
10. S.V. Gheorghita, T. Basten and H. Corporaal, "Intra-task Scenario-aware Voltage Scheduling," in *Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systesms (CASES)*, ACM Press, New York, NY, 2005, pp. 177–184.
11. S.V. Gheorghita, T. Basten and H. Corporaal, "Application Scenarios in Streaming-oriented Embedded System Design," in *Proc. of the International Symposium on System-on-Chip* (SoC 2006). IEEE Computer Society Press, Los Alamitos, CA, 2006, pp. 175–178.
12. S.V. Gheorghita, T. Basten and H. Corporaal, "Profiling Driven Scenario Detection and Prediction for Multimedia Applications," in *Proc. of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS)*, IEEE Computer Society Press, Los Alamitos, CA, 2006, pp. 63–70.
13. S.V. Gheorghita, S. Stuijk, T. Basten and H. Corporaal, "Automatic Scenario detection for improved WCET estimation, in *Proc. of the 42nd Design Automation Conference DAC*, ACM Press, New York, NY, 2005, pp. 101–104.

14. M. Hind, M. Burke, P. Carini and J. Choi, "Interprocedural Pointer Alias Analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, 1999, pp. 848–894.
15. Y. Huang, S. Chakraborty and Y. Wang, "Using Offline Bitstream Analysis for Power-aware Video Decoding in Portable Devices," in *Proc. of the 13th ACM International Conference on Multimedia*, ACM Press, New York, NY, 2005, pp. 299–302.
16. Intel Corporation: Intel XScale microarchitecture for the PXA255 processor: User's manual (2003). Order No. 278796.
17. N.K. Jha, "Low Power System Scheduling and Synthesis," in *Proc. of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, IEEE Computer Society Press, Los Alamitos, CA, 2001, pp. 259–263.
18. K. Lagerström, "Design and Implementation of an MP3 decoder. http://www.kmlager.com/mp3/. M.Sc. thesis, Chalmers University of Technology, Sweden, 2001.
19. A. Maxiaguine, Y. Liu, S. Chakraborty and W.T. Ooi, "Identifying 'representative' Workloads in Designing MpSoC Platforms for Media Processing," in *Proc. of 2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, IEEE Computer Society Press, Los Alamitos, CA, 2004, pp. 41–46.
20. E.J. McCluskey, "Minimization of Boolean Functions," *Bell Syst. Tech. J.*, vol. 35, no. 5, 1956, pp. 1417–1444.
21. MPEG Software Simulation Group, "MPEG-2 video codec," 2006. ftp://ftp.mpegtv.com/pub/mpeg/mssg/mpeg2vidcodec_v12.tar.gz.
22. S. Murali, M. Coenen, A. Radulescu, Goossens, K. and G. DeMicheli, "A Methodology for Mapping Multiple Use-cases Onto Networks on Chips," in *Proc. of Design, Automation, and Test in Europe* (DATE), IEEE Computer Society Press, Los Alamitos, CA, 2006, pp. 118–123.
23. M. Palkovic, H. Corporaal and F. Catthoor, "Global Memory Optimisation for Embedded Systems Allowed by Code Duplication," in *Proc. of the 9th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, ACM Press, New York, NY, 2005, pp. 72–79.
24. J.M. Paul, D.E. Thomas, and A. Bobrek, "Scenario-oriented Design for Single-chip Heterogeneous Multiprocessors, "*IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, 2006, pp. 868–880.
25. M. Pedram, W.C. Cheng, K. Dantu and K. Choi, "Frame-based Dynamic Voltage and Frequency Scaling for a MPEG decoder," in *Proc. of {IEEE/ACM} International Conference on Computer-Aided Design (ICCAD)*, ACM Press, New York, NY, 2002, pp. 732–737.
26. P. Poplavko, T. Basten, M. Pastrnak, J. van Meerbergen, M. Bekooij and P. de With, "Estimation of Execution Times of On-chip Multiprocessors Stream-oriented Applications," in *Proc. of the 3rd ACM/IEEE International Conference in Formal Methods and Models for Codesign (MEMOCODE)*, IEEE Computer Society Press, Los Alamitos, CA, 2005, pp. 251–252.
27. M.J. Rutten, J.T.J. van Eijndhoven, E.G.T. Jaspers, P. van der Wolf, E.D. Pol, O.P. Gangwal and A. Timmer, "A Heterogeneous Multiprocessor Architecture for Flexible Media Processing," *IEEE Des. Test Comput.*, vol. 19, no. 4, 2002, pp. 39–50.
28. R. Sasanka, C.J. Hughes and S.V. Adve, "Joint Local and Global Hardware Adaptations for Energy," *ACM SIGARCH Comput Archit News*, vol. 30, no. 5, 2002, pp. 144–155.
29. D. Shin and J. Kim, "Optimizing Intra-task Voltage Scheduling Using Data Flow Analysis," in *Proc. of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, ACM Press, New York, NY, 2005, pp. 703–708.
30. Sun Microsystems, Inc., Free Implementation of CCITT compression types G.711, G.721 and G.723, 2006.
31. Tektronix, "MPEG-2 video test bitstreams," 2006. ftp://ftp.tek.com/tv/test/streams/Element/MPEG-Video/525/.
32. I. Wegener, "Integer-Valued DDs," in *Branching Programs and Binary Decision Diagrams: Theory and Applications, SIAM Monographs on Discrete Mathematics and Applications*, chap. 9. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.
33. P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Cost-efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-time Multimedia Systems," in *Multi-Processor Systems on Chip*, chap. 11, W. Wolf, A. Jerraya (Eds.), Morgan Kaufmann, San Francisco, CA, 2003.

Valentin Gheorghita received his B.Sc. and M.Sc. degrees in Computer Science and Engineering from the ''Politehnica'' University of Bucharest, Romania in 2002, and respectively 2003. During his studies, he received two six-month research scholarships, one in the Tampere University of Technology, Finland (2000) and one in the National University of Singapore (2002). Moreover, he won multiple prizes at international programming contests, and he worked for three years in different software and consultancy companies. In 2003, Valentin Gheorghita joined the Electrical Engineering Department from the Eindhoven University of Technology, Netherlands for his Ph.D. studies. He is due to defend his Ph.D. in 2007. The current focus of his research is on embedded systems, especially on applications and design flow. During his Ph.D. studies, in the fall of 2005, he went for a three-month internship at Google

Inc., Mountain View, CA. Valentin Gheorghita published more than 10 scientific publications.

Dr.ir. Twan Basten is an Associate Professor in the Department of Electrical Engineering at the Eindhoven University of Technology. He has a Master's degree (with honors) and a Ph.D. degree in Computing Science from the same university. Twan Basten worked as visiting researcher at the University of Waterloo, Canada, Philips Research Laboratories, Eindhoven and Carnegie Mellon University, Pittsburgh, PA. His research interest is the design of complex, resource-constrained embedded systems, based on a solid mathematical foundation, with a focus on multiprocessor systems. Twan Basten was the Ambient Intelligence co-chair in the DATE 2003 PC, topic chair in the DATE 2004 and 2005 PCs, and the PC co-chair for ACSD 2007. He (co)authored over 80 scientific publications. He is a member of the ACM and a senior member of the IEEE.

Henk Corporaal has gained a M.Sc. in Theoretical Physics from the University of Groningen, and a Ph.D. in Electrical Engineering, in the area of Computer Architecture, from Delft University of Technology. Corporaal has been teaching at several schools for higher education, has been Associate Professor at the Delft University of Technology in the field of computer architecture and code generation, had a joint professor appointment at the National University of Singapore, and has been Scientific Director of the joined NUS-TUE Design Technology Institute. He also has been Department Head and Chief Scientist within the DESICS (Design Technology for Integrated Information and Communication Systems) division at IMEC, Leuven (Belgium). Currently, Corporaal is Professor in Embedded System Architectures at the Einhoven University of Technology (TU/e) in The Netherlands. He has co-authored over 200 journal and conference papers in the (multi-)processor architecture and embedded system design area. Furthermore, he invented a new class of VLIW architectures, the Transport Triggered Architectures, which is used in several commercial products. His current research projects are on the predictable design of soft- and hard real-time embedded systems.