Profiling Driven Scenario Detection and Prediction for Multimedia Applications*

Stefan Valentin Gheorghita, Twan Basten and Henk Corporaal
EE Department, Electronic Systems Group
Eindhoven University of Technology
PO Box 513, 5600 MB, Eindhoven, The Netherlands
{s.v.gheorghita,a.a.basten,h.corporaal}@tue.nl

***Abstract*— Modern multimedia applications usually have real-time constraints and they are implemented using heterogeneous multiprocessor systems-on-chip. 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, *scenarios*. This paper presents a method and a tool that can automatically detect the most important variables from an application and use them to define and dynamically predict scenarios, with respect to the necessary time budget, for soft real-time multimedia applications. The tool was tested for two multimedia applications. Using a proactive scenario-based scheduler based on the scenarios and the runtime predictor generated by our tool, the cycle budget over-estimation decreases with up to 83.50%, paying an acceptable cost of up to 1.74% in the number of missed deadlines.**

I. Introduction

Embedded systems usually consist of processors that execute domain-specific programs. Many of their functionalities are implemented in software, which is running on one or multiple generic 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 multimedia processing applications, like video and audio decoders. Because many of these systems are real-time portable embedded systems, they have strong requirements regarding size, performance and power consumption. The requirements may be expressed as: *the cheapest, smallest and most power efficient system that may deliver the required performance*. For dimensioning the system, accurate estimations of the resources needed by the application are required, like the number of execution cycles, memory-usage or 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 [1]. In order to save energy and still meet the real-time constraints of the multimedia applications, many power aware techniques based on dynamic voltage scaling (DVS) and dynamic power management (DPM) exploit this variability [2]. They scale the voltage and frequency of the processors at runtime to match the changing workload. Towards this, two main broad classes of techniques were involved: (i) *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 saved time and (ii) *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 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 [3], [4], or it may be encoded like meta-data into the input bitstream during an offline analysis [5], [6]. To avoid an explosion in the number of different cases that are considered and 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 resources (e.g. execution cycles) are combined together into, so called, *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 related to hard real-time systems presented in [7], [4]. In this paper, we describe *a method and a tool that can automatically detect the most important parameters and use them to define and dynamically predict scenarios for soft real-time multimedia applications*. This method extends our previous work, overcoming the limitations of the static analysis used in [7], but it can not be applied to hard real-time applications, as the scenario detection and prediction is not conservative. The quality of the results provided by our tool were tested for two applications: an MP3 decoder and the motion compensation task of an MPEG-2 decoder. For both of them, the set of parameters detected by our tool is similar to the one manually selected by the designer, and more complete

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

[1]The unused processor cycles represent the difference between how many cycles were reserved for a process and how many cycles were really needed.

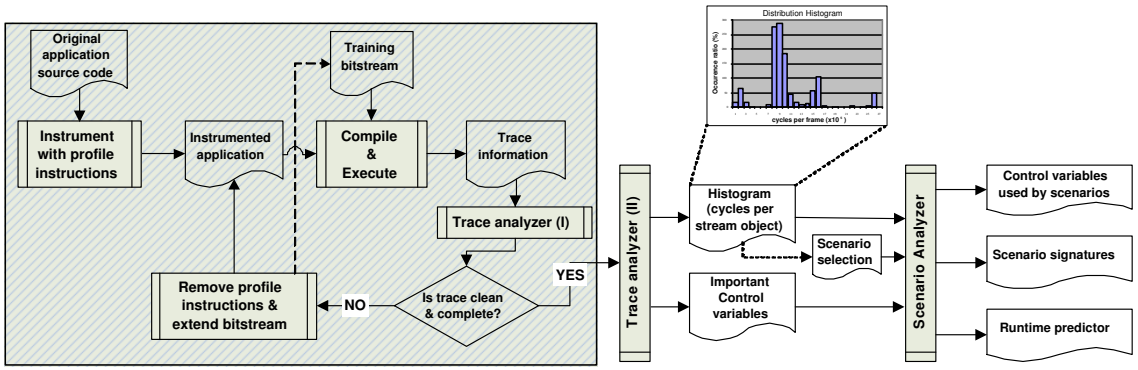

Fig. 1. Tool-flow for deriving parameters, scenarios and a runtime predictor.

than the one detected based on static analysis from [7]. Also, we show that using a scheduler that estimates the necessary cycle budget[2] for an application based on the scenarios and the runtime predictor generated by our tool, the total over-estimation for the application decreases substantially, at a cost of an acceptable penalty on the number of missed deadlines.

The paper is organized as follows. Section II shows different proactive power-aware approaches for saving energy for real-time systems, and presents how our current work is different. Section III describes our method and the developed tool-flow. In section IV, we evaluate our scenario detection and prediction method on two realistic multimedia decoders. Conclusions and future research are discussed in section V.

II. Related work

There are many approaches which propose to a-priori process the input bitstream 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 [5], [6], [8] 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, their approach uses 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 [3]. 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 [4] we proposed a way to control this overhead, 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 [4] 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 hardware unpredictability and WCEC analysis limitations. To overcome this issue, in the current paper, we use a profiling driven approach to detect and characterize scenarios. It solves the issue of manually detecting scenarios in the soft real-time frame-based dynamic voltage scaling algorithms, like the one presented in [9]. In addition, we present a technique to automatically derive and insert predictors in the application code to predict scenarios at runtime.

[2]This kind of scheduler is the basis of proactive DVS-aware schedulers. Based on the estimated cycle budget the DVS-aware schedulers adapt the processor voltage/frequency.

III. Methodology

This section starts with a presentation of the characteristics of multimedia applications and of the use of application parameters to estimate the necessary cycle budget. The remaining part of the section details each step of our method, of which overview is given in figure 1.

A. Multimedia applications

Many multimedia applications are implemented as a main loop that reads, decodes and writes out individual stream objects (see figure 2). A stream object might be a bit belonging to a compressed bitstream representing a coded video clip, a macro-block, a video frame, or an audio sample. 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 decode part consists of several kernels. For the decoding of each frame some of these kernels are used, depending on the frame type. The write part sends the decoded data to the output devices, like a screen or speakers, and saves

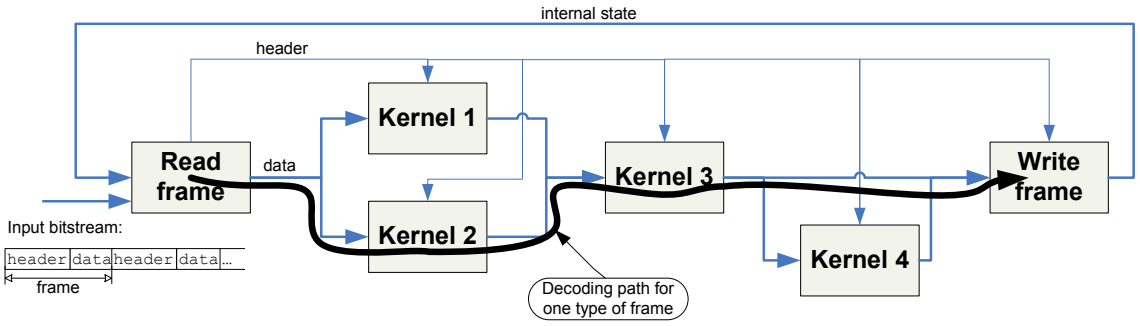

Fig. 2. Typical multimedia application decoding a frame.

the internal state of the decoder for further usage (e.g. in a video decoder, the previous decoded frame may be necessary to decode the current frame). Usually, these decoders have to deliver a given throughput (number of frames per second), which imposes a time constraint for each loop iteration.

B. Cycle budget estimation

For dimensioning a system, accurate estimations of the resources needed by the application to meet the desired throughput are required. For this paper we focus on the cycle budget needed to decode a frame in a specific period of time. This budget depends on the frame itself and the application internal state. In relevant related work, it is typically assumed that the budget $t(i)$ for frame i can be estimated using a linear function on data-dependent arguments with data-independent coefficients:

$$t(i) = C_0 + \sum_{k=1}^{n} C_k \xi_k(i), \qquad (1)$$

where the C_k are constant coefficients that depend on the processor type, and the $\xi_k(i)$ are n arguments that depend on the frame i from the input bitstream. Using for each frame its own transformation function with all possible source-code variables 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 time are treated together, being combined in so called *scenarios*. To reduce the size of each function, only the variables whose values influence the decoding time of a frame the most should be used.

C. Control variable identification

The variables that appear in an application may be divided in *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 iterations 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. We can easily observe 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 decoding time for 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. Both aspects are handled by the trace analyzer discussed in the next subsection.

The large gray box from figure 1 shows the work-flow for control variable identification. It starts from the source code of the multimedia application, which is then instrumented with profile instructions for all read and write operations to the variables. The instrumented code is then 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 [10], where the authors show a technique for classifying different multimedia streams, could be used. The analysis done 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 any data variable. As our method generates a huge trace if it is applied from the beginning on a large bitstream, we start with a few frames 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 program trace does not contain any data variable anymore.

D. Trace analyzer

The trace analyzer has two roles: (i) 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 (ii) when the process is complete, it generates the data necessary for the scenario selection step

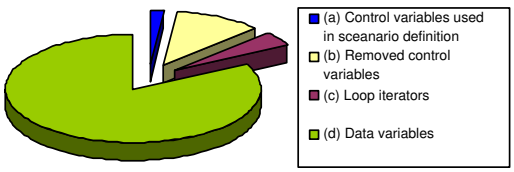

Fig. 3. Variable distribution for MP3.

explained in subsection III-E and a list of the remaining control variables.

To identify the data variables stored in large implicitly declared arrays, that can not be detected based on static analysis (e.g. an array that appears in the source code as a pointer), the trace analyzer applies the following rule: if in the trace information generated for each frame, there is a program operation that reads or writes a number of different memory addresses larger than a threshold, we consider that all these memory addresses are linked to data variables, as this operation looks like accessing an array. For this decision we do not look for a specific array access pattern. 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.

The control variables which are considered to have a small influence on the application execution time and that are easy to identify based on the trace information generated for each frame are the loop iterators. 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 `while (i<n) i++;`, the variable `n` bounds the number of iterations, while the loop iterator `i` counts them. If there is a program operation that writes more than once the same variable, this variable can be considered a loop iterator[3].

After the process 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 equation 1. In the following, we present a way to further reduce their number. Figure 3 shows the categories in which the application variables are divided.

Besides the write and read operations, the program trace contains also the number of cycles needed to decode each frame. This information is used in the scenario selection step. The trace analyzer depicts them graphically as a distribution histogram, showing 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.

E. Scenario selection

The distribution histogram is the input of a heuristic algorithm for splitting executions into scenarios. Each scenario j is characterized by a cycle count interval

$$(cycles_{lb}(j), cycles_{ub}(j)] \tag{2}$$

that bounds the number of cycles needed to decode each frame that is part of the scenario.

An example of a distribution histogram and scenario selection based on it is presented in figure 5, discussed in detail in section IV-A. For this example, based on the lines marked with `I`, the application behavior is split into two scenarios, namely: $(1.1 \cdot 10^6, 1.9 \cdot 10^6]$ and $(1.9 \cdot 10^6, 3.8 \cdot 10^6]$. This scenario selection step is the only manual step in our work; automating it is an open point for future research. However, since it is based on visual information, this is relatively straightforward for a designer. The splitting depends on how the scenarios are used. For example, when they are used to adapt processor voltage/frequency in advance based on the cycle budget needed for decoding a frame, the following criteria can be used as the heuristic:

- Immediately on the right of each peak in the distribution histogram, or a group of close peaks, an upper bound of a scenario is selected. This is because for the frames that need a cycle budget close to the upper bound of the interval that characterizes a scenario, the over-reservation is smaller than for the ones that need a budget close to the lower bound.
- Depending on how expensive, in cycles, it is to reconfigure the system (i.e. scale voltage/frequency) at runtime, the number of possible scenario transitions, for which statistics can be derived from the trace, may be taken into account.
- The granularity of possible system configurations (voltage/frequency levels) must be considered.
- The possible runtime prediction error and the cost of prediction must be taken into account (see the next section).

F. Scenario analyzer

The scenario analyzer step from figure 1 takes into account the generated trace for the entire training bitstream and the above scenario selection. It generates: (i) for each scenario, an equation that characterizes the scenario depending on the application control variables; (ii) a list of the most important parameters from the application; and (iii) the source code of the predictor that can be used to predict at runtime in which scenario the application is running.

Frame signature: The analyzer starts by deriving for each frame from the training bitstream its signature. It is obtained by processing the generated trace and it is defined as a pair:

$$(s(i), \{(\xi_k, v_k(i), op_k(i)) | \xi_k \ read \ or \ written\}), \tag{3}$$

where $s(i)$ is an identifier defining the scenario to which frame i is associated, ξ_k are the control variables, $v_k(i)$ is the value of the variable ξ_k in frame i and $op_k(i)$ is the instruction from the source code that is the last, considering the execution time

[3]The same behavior appears also in the case of counters, but we do not make the difference between them and iterators, removing the variables in both cases.

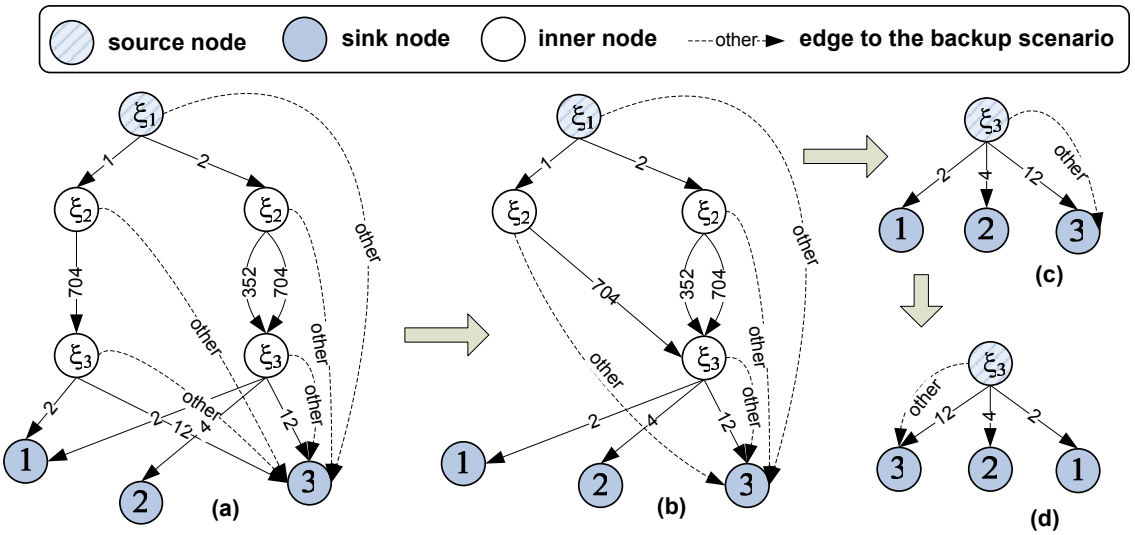

Fig. 4. Simplified motion compensation decision diagrams: (a) original; (b) and (c) reduced; (d) reordered.

line, to change the value of ξ_k. If ξ_k is only read when frame i is decoded, $op_k(i) = nil$.

Scenario equation: For each frame i, using its signature, a boolean function $\chi_f(i)$ over variables ξ_k characterizing the frame is defined

$$\chi_f(i)(\overrightarrow{\xi_k}) = \bigwedge_k (\xi_k = v_k(i)). \tag{4}$$

By using these functions, for each scenario j, a boolean function $\chi_s(j)$ over variables ξ_k characterizing the scenario is defined:

$$\chi_s(j)(\overrightarrow{\xi_k}) = \bigvee_{i,s(i)=j} \chi_f(i)(\overrightarrow{\xi_k}). \tag{5}$$

The canonical form of this boolean function is obtained using the Quine McCluskey algorithm [11]. The idea is that precisely 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 and only over control variables, two special cases may appear when a new frame i is checked against them:

- **Conflict:** There are two or more scenarios j for which $\chi_s(j)(\overrightarrow{v_k(i)})$ evaluates to $true$. In this case, to be on the safe side, the frame is cataloged to be in the scenario j that has the largest value $cycles_{ub}(j)$ among those for which the characteristic function evaluates to $true$.
- **Not found:** There is no scenario j for which $\chi_s(j)(\overrightarrow{v_k(i)})$ evaluates to $true$. In this case the frame is cataloged to be in the so-called *backup scenario* b, which is the scenario j with the largest $cycles_{ub}(j)$ 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 on all possible paths within the main loop of the multimedia application the instruction that is the last to change the value of any variable ξ_k is identified. After each of these instructions an identical runtime predictor is inserted. 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 the following will refine the set until only one scenario remains. However, we leave this point open for future research.

The runtime predictor appears as a *multi-valued decision diagram* [12], defining a function

$$f : \Omega_1 \times \Omega_2 \times ... \times \Omega_n \rightarrow \{1, .., m\}, \tag{6}$$

where Ω_k is the set of all possible values of the type of variable ξ_k 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 $v_k(i)$ 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 $v_k(i)$ that appear for variable ξ_k in all frames from the training bitstream plus an edge labeled with *other* that leads directly to the backup scenario. There is only one inner node without incoming edges in V which is the source node of the diagram, 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 is shown in figure 4(a). As the information used to build the decision diagrams is the same as the one used to define scenario characterization functions, both special cases presented above, *conflict* and *not found* appear. They are solved during the decision diagram construction.

When the decision diagram is used in the source code to predict the future scenario, it introduces two additional

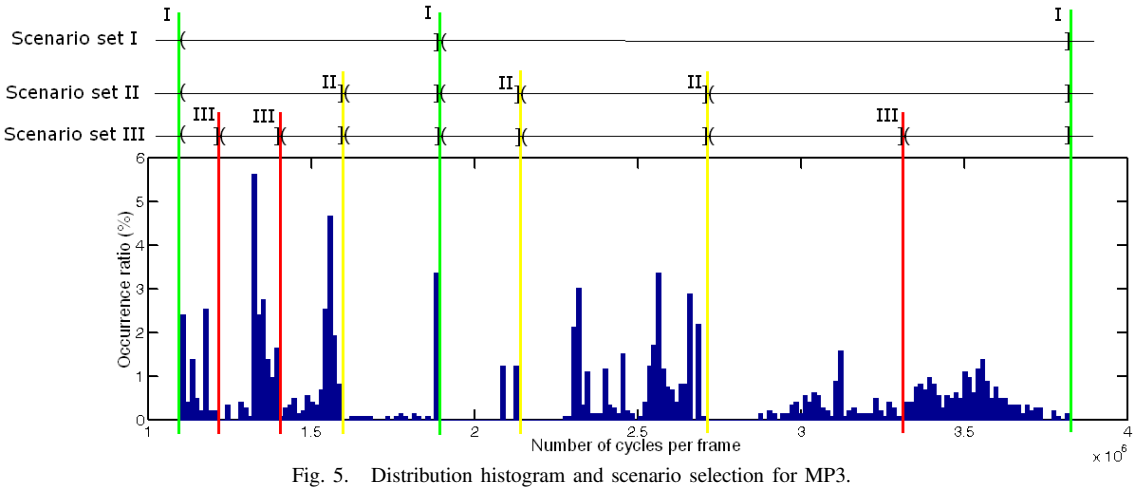

Fig. 5. Distribution histogram and scenario selection for MP3.

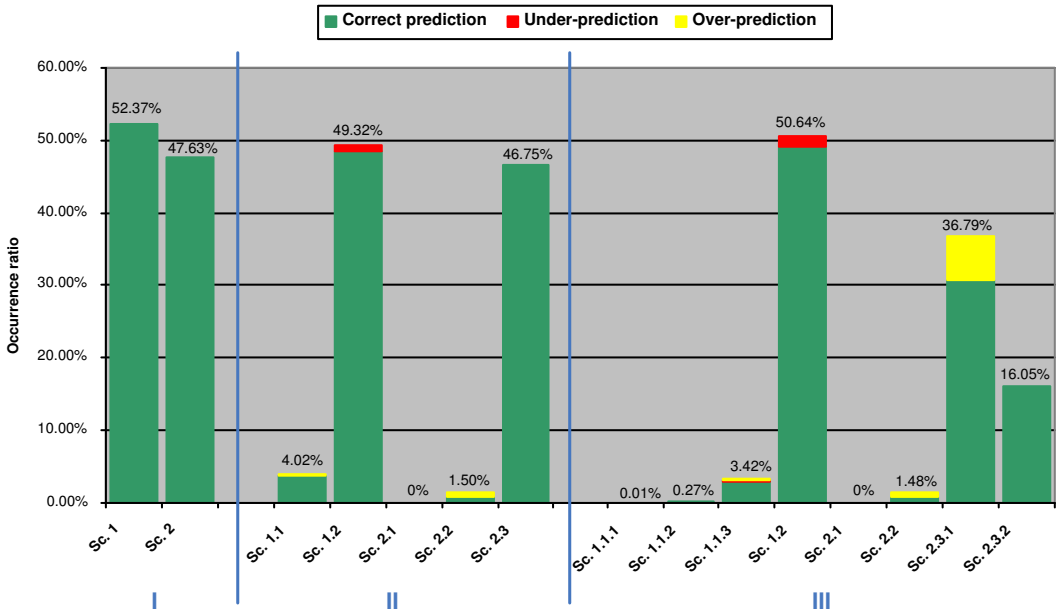

Fig. 6. Scenario prediction for MP3.

cost factors: (i) *decision diagram code size* and (ii) *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. For example, for the decision diagram from figure 4(a), it can be assumed that if $\xi_1 = 1$, $\xi_2 = 704$ and $\xi_3 = 4$ the application is, probably, in scenario 2. This case did not appear for the training bitstream and the assumption is based on the rest of the diagram structure. If this assumption is made, the decision diagram can be reduced to the one from figure 4(b). 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. This means that we can remove the variables ξ_1 and ξ_2 from the diagram (see figure 4(c)), with the risk that, if the values of ξ_1 and ξ_2 for a frame did not appear in the training bitstream, a scenario is selected instead of the conservative backup scenario. To decrease the average evaluation cost, the outgoing edges of each inner node from V are sorted in descending order based on the occurrence ratio of the values that label them. In figure 4(d), the edges for the node labeled with ξ_3 were reordered, considering the case when $\xi_3 = 12$ appears most often. Our tool produces two decision diagrams, the original one created only based on the training bitstream and the one on which all possible size reductions were applied. The basic rule governing the reductions is that they should not introduce conflicts. Both diagrams are ordered to optimize the average evaluation cost. The designer can choose which one to use; by default the tool uses the reduced one as the predictor.

Most important control variables: In the steps presented above, the data variables and the loop iterators were removed (fig. 3). From the remaining variables, we select the ones that influence the execution time the most, as the variables used in the decision diagram selected by the designer. To be on the safe side when the training bitstream was not *representative* enough, we complete the set with the variables with large

Scenario set	I	II	III
Design time			
Number of scenarios	2	5	8
Decision diagram size (comparisons)	3	68	256
Average prediction cost (comparisons)	2.41	6.17	11.58
Runtime			
Under-prediction	0%	0.74%	1.74%
Over-prediction	0%	0.85%	6.92%
Deadline misses with a one-frame output buffer	0	0	0
Average cycle budget over-estimation reduction (all files)	71.65%	72%	83.50%
Average cycle budget over-estimation reduction (stereo files)	0%	1%	46.3%

TABLE I

EVALUATION OF SCENARIO SETS FOR MP3

influence on the application WCEC, as detected by the static analysis presented in [7].

G. Tool-flow

All the steps of the presented tool-flow, except scenario selection, were implemented on top of SUIF [13], and work for applications written in C, as C is the most used language to write embedded systems software.

IV. EXPERIMENTAL RESULTS

We tested our method on two multimedia applications, an MP3 decoder [14] and the motion compensation task of an MPEG-2 decoder. For both applications we show how our tool can help in splitting the application into scenarios that are used to reduce the cycle budget over-estimation, leading to a smaller and energy efficient system. The generated runtime predictor introduces a low rate of deadline misses, that is usually acceptable in case of soft real-time applications.

A. MP3 Decoder

The MPEG-I Layer III decoder is a frame-based algorithm, which transforms the compressed bitstream in normal pulse code modulation data. A frame consists of 1152 mono or stereo frequency-domain samples, divided into two granules. Details about the application structure and the source code are presented in [14]. To profile the application, we have chosen, as the training bitstream, a set of audio files consisting of: (i) the ones taken from [15], which were designed to cover all the extreme cases, and (ii) a few randomly selected stereo and mono songs downloaded from the internet, in order to cover the common cases. After removing the data variables and loop iterators, the number of remaining variables ξ_k is 41. This set of variables is far more complete than the one detected using the static analysis from [7].

The distribution histogram for the given training bitstream together with the steps done to split into scenarios is shown in figure 5. The splitting into scenarios is done in three different iterations. Except for the first iteration, when we start from the original application, for each of the rest, we start from the previously selected set of scenarios.

In the first step, we take into account that the decoder works on both stereo and mono frames. Considering this, we assume that the amount of computation is double for stereo, compared to mono frames. Based on this observation, we split the application into two scenarios, as shown by the lines marked with `I` in figure 5. In the second step (marked with `II`), we intuitively split in a way that most of the cycle budgets that did not or not often appear in the training bitstream are close to the lower bound of the intervals that characterize the scenarios. In the third iteration, we select the upper bounds of scenario intervals immediately after a peak or a group of peaks.

For each set of scenarios, an evaluation of the ratio of occurrence of each scenario and its under- and over-prediction, for randomly selected stereo and mono input files (different from the ones from the training set) is presented in figure 6. The prediction errors appear as not all possible situations were covered by the training bitstream. Using a scheduler that reserves the cycle budget for the application based on the estimated one (e.g. a proactive DVS-aware scheduler), the under-predictions would correspond to deadline misses. The scenario over-predictions do not influence the number of deadline misses; they lead only to over-estimation. The more scenarios, the larger the prediction errors, but as we will see also, the larger potential for reducing the frame average cycle budget over-estimation.

Table I shows, for each set of scenarios, the cost of the prediction, given by the decision diagram code size, and average runtime. The cost is given in number of comparisons. Considering that the code size for a comparison is less than 10 words and the runtime evaluation less than 10 cycles, both costs are negligible with respect to the original application. In order to evaluate the quality of each scenario set, we look at reduction in cycle budget over-estimation, for all mono and stereo input files, and separately only for stereo files[4], considering as the reference for comparison the case when the entire application behaves like one scenario. Observe that the scenario approach leads to an under-prediction of necessary cycle budget up to 1.74%. We can solve this problem by using an output buffer that can store one frame. In this case, the number of deadline misses is reduced to 0 for all three sets of scenarios.

It is clear that scenario set `III` has a large potential for energy saving using a DVS system, even when only stereo files are considered, at the cost of 1.74% of deadline misses or 0% of deadline misses with a frame output buffer. In future work we plan to quantify potential energy saving using a DVS

[4]In most cases, users listen only to stereo music.

Scenario set	I	II
Number of scenarios	2	5
Decision diagram size	12	38
Average prediction cost	3.41	7.37
Deadline misses ratio	0%	0.001%
Avg. over-estimation reduction	69.91%	74.23%

TABLE II

EVALUATION OF SCENARIO SETS FOR MC

system, taking into account all aspects like frequency/voltage adaptation costs, and time and energy overhead of the predictors.

B. MPEG-2 Motion Compensation

An MPEG-2 video sequence is composed by 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 discr ete 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 [16], and as the training bitstream we consider the first 10000 MBs from each test file from [17]. As IDCT execution time for each MB is almost constant, we focus on MC and VLD. In case of 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 [8], and which can be found in the source code. Using our method to split into scenarios, and considering the entire bitstreams from [17] as the input streams, we obtained a 74.23% reduction in cycle budget over-estimation paying a cost of a 0.001% miss-ratio. Table II presents a full evaluation of the selected sets of scenarios.

V. CONCLUSION

In this paper we have presented a profiling driven approach to detect and characterize scenarios for 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 to predict scenarios at runtime. Our method was tested for two multimedia applications. We show that using a proactive scheduler based on the scenarios and the runtime predictor generated by our tool, the cycle budget over-estimation decreases with 83.50%, respectively 74.23%, paying an acceptable cost of 1.74%, respectively 0.001%, in the number of missed deadlines. We show that, if the miss deadline ratio is not acceptable, a frame output buffer can be used to reduce it.

In future work we plan to explore different ways of automatically selecting scenarios and to quantify potential energy saving using a DVS system, taking into account all aspects like frequency/voltage adaptation costs and time and energy overhead of the predictors. Moreover, we would like to investigate solutions for error recovery in case of under-prediction errors and to analyze the quality and the runtime cost of using adaptable predictors, i.e., predictors that adapt their scenario definition and decision diagram at runtime.

REFERENCES

[1] M. J. Rutten, J. T. J. van Eijndhoven, E. G. T. Jaspers, P. van der Wolf, E.-J. D. Pol, O. P. Gangwal, and A. Timmer, "A heterogeneous multiprocessor architecture for flexible media processing," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 39–50, July 2002.

[2] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, San Jose, CA, USA, November 2001, pp. 259–263.

[3] 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*. Shanghai, China: IEEE, January 2005.

[4] 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 (CASES2005)*. San Francisco, CA, USA: ACM Press, NY, USA, September 2005, pp. 177–184.

[5] A. C. Bavier, A. B. Montz, and L. L. Peterson, "Predicting MPEG execution times," *ACM SIGMETRICS Performance Evaluation Review archive*, vol. 26, no. 1, pp. 131–140, June 1998.

[6] 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 2005)*, Verona, Italy, July 2005, pp. 251–252.

[7] 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*. Anaheim, CA, USA: ACM Press, NY, USA, June 2005, pp. 101–104.

[8] 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*, Singapore, November 2005, pp. 299–302.

[9] M. Pedram, W.-C. Cheng, K. Dantu, and K. Choi, "Frame-based dynamic voltage and frequency scaling for a MPEG decoder," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '02)*, San Jose, CA, USA, November 2002, pp. 732–737.

[10] 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)*. Stockholm, Sweden: IEEE, September 2004.

[11] E. McCluskey, "Minimization of boolean functions," *Bell System Technical Journal*, vol. 35, no. 5, pp. 1417–1444, 1956.

[12] I. Wegener, "Integer-Valued DDs," in *Branching Programs and Binary Decision Diagrams:Theory and Applications*, ser. SIAM Monographs on Discrete Mathematics and Applications. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2000, ch. 9.

[13] S. Amarasinghe, J. Anderson, M. Lam, and C. W. Tseng, "An overview of the SUIF compiler for scalable parallel machines," in *Proc. of the 7th Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995.

[14] K. Lagerström, "Design and implementation of an MP3 decoder," May 2001, m.Sc. thesis, Chalmers University of Technology, Sweden. [Online]. Available: http://www.kmlager.com/mp3/

[15] M. Dietz and et al., "MPEG-1 audio layer III test bitstream package," May 1994, http://www.iis.fhg.de.

[16] M. S. S. Group, "MPEG-2 video codec," ftp://ftp.mpegtv.com/pub/mpeg/mssg/mpeg2vidcodec_v12.tar.gz.

[17] Tektronix, "MPEG-2 video test bitstreams," ftp://ftp.tek.com/tv/test/streams/Element/MPEG-Video/525/.