Technische Universiteit
Eindhoven
University of Technology

System Scenario based Design of Dynamic Embedded Systems

S. V. GHEORGHITA, M. PALKOVIC, J. HAMERS, A. VANDECAPPELLE,
S. MAMAGKAKIS, T. BASTEN, L. EECKHOUT, H. CORPORAAL,
F. CATTHOOR, F. VANDEPUTTE and K. De BOSSCHERE

ES Reports
ISSN 1574-9517

ESR-2007-06
30 September 2007, revised 23 June 2008

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

© 2007, 2008 Technische Universiteit Eindhoven, Electronic Systems.
All rights reserved.

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

System Scenario based Design of Dynamic Embedded Systems

S. V. GHEORGHITA[1], M. PALKOVIC[2], J. HAMERS[3], A. VANDECAPPELLE[2], S. MAMAGKAKIS[2], T. BASTEN[1], L. EECKHOUT[3], H. CORPORAAL[1], F. CATTHOOR[2,4], F. VANDEPUTTE[3] and K. De BOSSCHERE[3]

[1] Eindhoven University of Technology, The Netherlands
{s.v.gheorghita,a.a.basten,h.corporaal}@tue.nl

[2] IMEC vzw, Leuven, Belgium
{palkovic,vdcappel,mamagka,catthoor}@imec.be

[3] Ghent University, Belgium
{juan.hamers,lieven.eeckhout,frederik.vandeputte,koen.debosschere}@elis.ugent.be

[4] Katholieke Universiteit Leuven, Belgium

http://www.es.ele.tue.nl/scenarios

In the past decade, real-time embedded systems have become much more complex due to the introduction of a lot of new functionality in one application, and due to running multiple applications concurrently. This increases the dynamic nature of today's applications and systems, and tightens the requirements for their constraints in terms of deadlines and energy consumption. State-of-the-art design methodologies try to cope with these novel issues by identifying several most used cases and dealing with them separately, reducing the newly introduced complexity. This paper presents a generic and systematic design-time/run-time methodology for handling the dynamic nature of modern embedded systems, which can be utilized by existing design methodologies to increase their efficiency. It is based on the concept of *system scenarios*, which group system behaviors that are similar from a multi-dimensional cost perspective, such as resource requirements, delay, and energy consumption, in such a way that the system can be configured to exploit this cost similarity. At design-time, these scenarios are individually optimized. Mechanisms for predicting the current scenario at run-time and for switching between scenarios are also derived. This design trajectory is augmented with a run-time calibration mechanism, which allows the system to learn on-the-fly during its execution, and to adapt itself to the current input stimuli, by extending the scenario set, changing the scenario definitions, and both the prediction and switching mechanisms. To show the generality of our methodology, we show how it has been applied on four very different real-life design problems. In all presented case studies substantial energy reductions were obtained by exploiting scenarios.

Categories and Subject Descriptors: C.3.d [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.2.10 [**Software Engineering**]: Design—*Methodologies*

This paper will appear in ACM Transactions on Design Automation of Electronic Systems (ToDAES).

1. INTRODUCTION

Embedded systems usually consist of processors that execute domain-specific applications. These systems are software intensive[1], having much of their functionality implemented in software, which is running on one or multiple processors, leaving only the high performance functions implemented in hardware. Typical examples include TV sets, cellular phones, wireless access points, MP3 players and printers. Most of these systems are running multimedia and/or telecom applications and support multiple standards. Thus, these applications are full of dynamism, i.e., their execution costs (e.g., number of processor cycles, memory usage, energy) are environment dependent (e.g., input data, processor temperature).

Scenario-based design [Carroll 1995] has been used for some time in both hardware [Ionita 2005; Paul et al. 2006] and software design [Douglass 2004] of embedded systems. In both these cases, scenarios concretely describe, in an early phase of the development process, the use of a future system. They appear like narrative descriptions of envisioned usage episodes, or unified modeling language (UML) use-case diagrams [Fowler 2003] which enumerate, from a functional and timing point of view, all possible user actions and the system reactions that are required to meet a proposed system function. These scenarios are called *use-case scenarios*. They focus on the application functional and timing behaviors and on its interaction with the users and environment, and not on the resources required by a system to meet its constraints. These scenarios are used as an input for design approaches centered around the application context.

In this paper, we concentrate on a different and complementary type of scenarios, which we call *system scenarios*. These are derived from the combination of the behavior of the application and the application mapping on the system platform. These scenarios are used to reduce the system cost by exploiting information about what can happen at run-time to make better design decisions at design-time, and to exploit the time-varying behavior at run-time. *While use-case scenarios classify the application's behavior based on the different ways the system can be used in its overall context, system scenarios classify the behavior based on the multi-dimensional cost tradeoff during the implementation trajectory.* By optimizing the system per scenario and by making sure that the actual system scenario can be predicted at run-time, a system setting can be derived per scenario to optimally exploit the system scenario knowledge.

To motivate the system scenario usage in embedded system design, we start from the different *Run-Time Situations* (RTSs) in which a system may run on a given system platform. An RTS is a piece of system execution that is treated as an unit. Each RTS has an associated cost, which usually consists of one or several primary costs, like quality and resource usage (e.g., number of processor cycles, memory size). The system execution is a sequence of RTSs, and the current RTS is known only at the moment it occurs. However, at run-time, using various system parameters, so-called *RTS parameters*, it can be predicted in advance in which RTS the system will run next for a non-zero future time window. If the information

[1]If the system's software contributes with essential elements to the design, construction, deployment, and evolution of the system as a whole, we talk about a software intensive system [IEEE 2000].

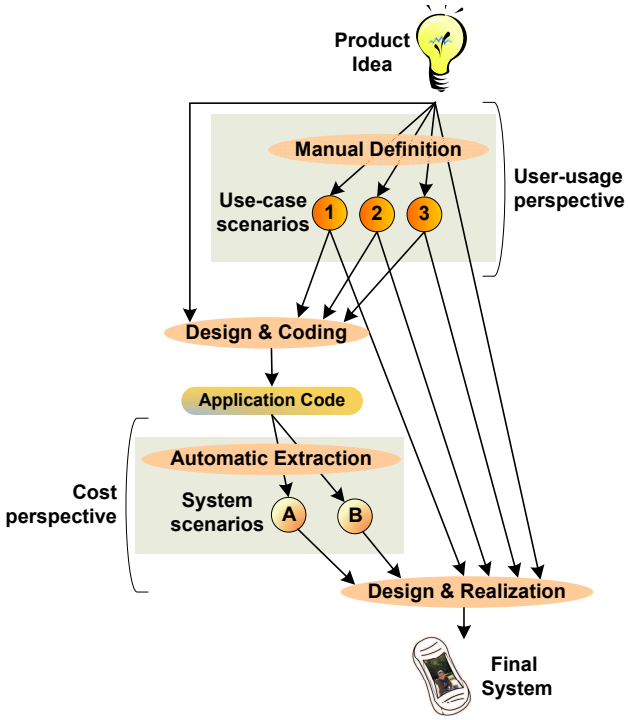

Fig. 1. A scenario based design flow for embedded systems.

about all possible RTSs in which a system may run is known at design-time, and the RTSs are considered in different steps of the embedded system design, a better optimized (e.g., faster or more energy efficient) system can be built because specific and aggressive design decisions can be made for each RTS. These intermediate per-RTS optimizations lead to *a smaller, cheaper and more energy efficient system that can deliver the required quality*. In general, any combination of N cost dimensions may be targeted. However, the number of cost dimensions and all possible values of the considered RTS parameters may lead to an exponential number of RTSs. This will degenerate to a long, and really complicated design process, that does not deliver the optimal system. Moreover, the run-time overhead of detecting all these different RTSs will be too high compared to the expected gain over their (quite) short time windows. To avoid this situation, in our work, the RTSs are classified and clustered from an N-dimensional cost perspective into *system scenarios*, such that the cost tradeoff combinations within a scenario are always fairly similar, the RTS parameter values allow an accurate prediction, and a system setting can be defined that allows to exploit the scenario knowledge and optimizations. This paper presents a systematic way of detecting and exploiting both at design-time and run-time the system scenarios of a given system. Generic solutions to the various steps in the methodology are provided whenever available. The method combines design-time analyses and optimizations with information collected at run-time about the environment in which the system is operating and the inputs being received.

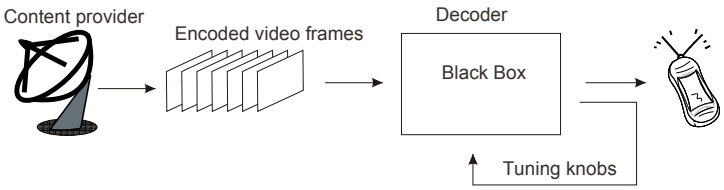

Fig. 2. Motivating example, no scenarios.

Figure 1 depicts a design trajectory using use-case and system scenarios. It starts from a product idea, for which the stakeholders[2] manually define the product's functionality as use-case scenarios. These scenarios characterize the system from a user perspective and are used as an input to the design of an embedded system that includes both software and hardware components. In order to optimize the design of the system, the detection and usage of system scenarios augments this trajectory (the bottom gray box from the figure). The run-time behavior of the system is classified using the methodology presented in this paper into several system scenarios, with similar cost tradeoffs within a scenario. For each individual scenario, more specific and aggressive design decisions can be made. The sets of use-case scenarios and system scenarios are not necessarily disjoint, and it is possible that one or more use-case scenarios correspond to one system scenario. But still, usually they are not overlapping and it is likely that a use-case scenario is split into several system scenarios, or even that several system scenarios intersect several use-case scenarios.

The paper is organized as follows. Section 2 gives a motivating example for our work, by showing how system scenario exploitation makes an H.264 video decoder more energy efficient. The system scenario methodology for embedded system design is detailed in section 3. It is accompanied in section 4 by case studies, that describe a diversity of applications that fit within this methodology, illustrating its broad application potential. Related work is presented in section 5, and the conclusions and our future plans are detailed in section 6.

2. MOTIVATING EXAMPLE

Figure 2 shows a typical system to which the system scenario design methodology is applicable. In this system, a content provider sends H.264 encoded sequences of video frames to a mobile device that decodes the content. The video decoder is often implemented as a main loop which is executed over and over again, reading encoded frames, decoding them and writing them to an output device (e.g., a screen). The application has to deliver a certain throughput (e.g., 30 frames per second), which imposes a time constraint on each loop iteration. Otherwise, the movie will stutter and the user's experience will degrade. When this kind of video decoder is implemented in a mobile device that is battery-operated and thus energy-constrained, the goal of using system scenarios when designing this system is to reduce the energy consumption, while retaining an acceptable frame rate.

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

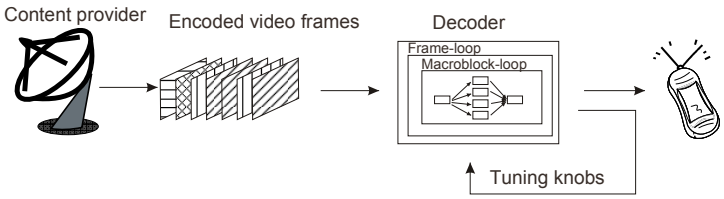

Fig. 3. Motivating example, system scenarios.

While every frame must be decoded within a fixed period of time, the actual time and energy needed to decode a frame on a processor with a given speed varies due to the dynamism exhibited by the video stream. Some frames require all the available decoding time while others demand only a fraction and leave the processor idle for the remaining time. On a small set of video sequences, we already noticed differences up to 450% in the required energy for decoding a single frame.

One well known technique for reducing energy consumption in this situation (fixed deadline, varying decode time) is Dynamic Voltage and Frequency Scaling (DVFS) [Jha 2001]. When scaling the voltage, the processor's frequency and therefore the execution time scales linearly ($f_{CLK} \propto V_{DD}$), while the energy consumption scales approximately quadratically ($E \propto V_{DD}^2$). As such, DVFS gives the system a *knob*, namely a choice to work at a certain frequency/voltage, that needs to be tuned at run-time. An important problem when applying DVFS in this situation is the need of knowing how many cycles (the *cycle budget*) are needed for decoding a frame, before actually decoding it. This is necessary to choose the appropriate scaling factor, i.e., to choose in which position to turn the DVFS system knob.

Existing DVFS systems work either fully dynamically based on run-time information, or fully statically based on design-time analysis. In the fully dynamic approach, no information about the decoder is considered except the notion of consecutive frames with different decode times that need to be decoded within a given deadline (figure 2). Without any information on how the internals of the decoder cause this variation in decode time, it is only possible to predict at run-time the required cycle budget of the current frame based upon the cycle budget needed for previously decoded frames [Hughes et al. 2001; Choi et al. 2002]. Another, fully static, approach considers complete information about the platform usage collected at design-time and uses static analysis to determine the remaining worst case cycle budget needed to complete execution at several points in the execution and fixes the DVFS system to the corresponding voltage and frequency [Shin and Kim 2005].

When using our proposed system scenarios (figure 3), we consider both information about the system at design-time, as well as the occurrence of certain types of input at run-time, which result in particular (groupings of) RTSs. Looking at the general structure of the H.264 decoder (figure 4), we see that each frame is subdivided into blocks of 16 by 16 pixels, called macroblocks. The main loop, which is the frame decoding loop, contains a second loop that iterates over the consecutive macroblocks. The read part of this loop takes an encoded macroblock object from the input stream and separates it into a *header* and the object's *data*. The write part places the decoded macroblock in the frame. The decoding part consists of several kernels. Each macroblock can be encoded using a different encoding scheme

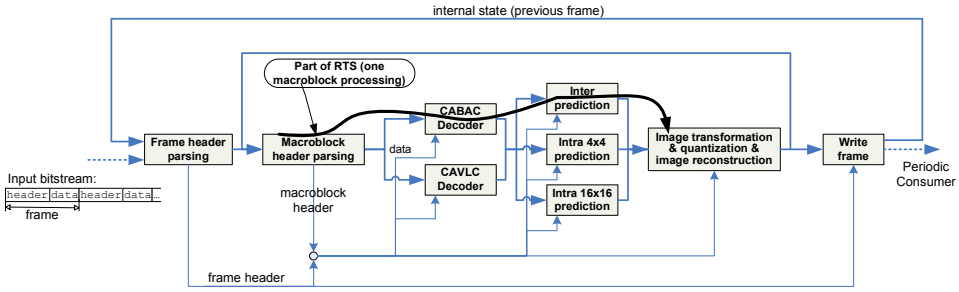

Fig. 4. Grey-box model of the H.264/AVC decoder processing a stream object.

that determines which kernels are used. Depending on the exact breakup of how many macroblocks of each frame belong to each scheme for a given iteration of the main frame loop, each of these kernels are executed for a certain number of times. This forms the *Run-Time Situation (RTS)* which can be characterized by the current *RTS parameters*, i.e., the macroblock breakup in this example. These parameters can be used to predict in advance the costs associated with the current RTS.

Considering each possible breakup of a frame for tuning the system at run-time would cause a too large overhead. When decoding CIF images (352 by 288 pixels), consisting of 396 macroblocks that may belong to 21 different encoding schemes, up to $6.22 \cdot 10^{33}$ possible RTSs would need to be considered, and for each RTS the client has to store the optimal frequency/voltage. This is clearly impossible. Therefore, it is necessary to cluster frames with a similar breakup of macroblock types over the encoding schemes and mapping on the target platform. So frames that need similar cycle budgets are merged into the same system scenario. For each system scenario, we then determine the optimal knob setting of the mapping scheme and the platform. For example, we can determine the frequency/voltage setting of the DVFS scheme by using the cycle budget needed for a single (worst case) representative frame as the budget needed for all possible frames belonging to this scenario.

At run-time, whenever a client receives a movie from a content provider, it predicts for each frame the scenario it belongs to. Then, it scales the voltage and frequency according to the values determined at design-time, thereby reducing the energy consumption while still meeting the deadlines. Clearly, the more scenarios are considered, the higher the energy reduction that can be obtained, but also the more complex the prediction becomes. The prediction causes run-time overhead and it will add to the energy usage. The cost and gain of extra scenarios have to be traded off carefully to arrive at an optimal system. In the extreme, one can select a pure run-time controller approach. But then either the run-time decision overhead becomes too high, or the quality of the decision becomes very poor due to the too local view. The scenario approach avoids this bad tradeoff point.

Another interesting issue is to what extent scenario prediction can be made or needs to be made conservative. For complexity reasons, it may not be possible to consider all possible RTSs in the scenario definition. For the H.264 decoder, for

example, there are too many RTSs to take them all into account explicitly. When at run-time a frame arrives with a previously unseen macroblock breakup, it needs to be decided what to do. Hard guarantees on system performance can be given by predicting for these new breakups the *backup scenario*, which is the scenario that needs the overall worst case number of cycles to execute. However, this will often lead to less energy reduction than potentially possible. Since for video decoding a small percentage of missed frame deadlines is usually acceptable, one could aim for a more aggressive prediction, introducing the risk that a frame may get miss-predicted as belonging to a scenario which has a lower cycle budget than the frame really needs, causing a missed deadline. This leads to a tradeoff between system quality in terms of missed deadlines and energy savings. In [Hamers et al. 2007], we managed to reduce energy consumption of the H.264 decoder by 46% with less than 0.1% of the deadlines missed, by using only 32 scenarios.

To exemplify the difference between use-case and system scenarios, let us consider a mobile device running an H.264 decoder that supports two different frame resolutions. From the user perspective, each resolution may be considered as a use-case scenario, because the resolution affects the perceived quality. Due to the different resolution, the two use-case scenarios contain a different number of macroblocks. Each of the two use-case scenarios can be divided automatically in more system scenarios based on the frame's macroblock mapping breakup, as presented above. This breakup is of no interest to the application designer or final product user, because it is a system-internal artefact of the video encoding, but it can be exploited to reduce energy consumption in the mapping. It may even be possible to integrate certain macroblock breakups of the two different resolutions into a single system scenario.

The following section details the systematic aspects of our methodology of identifying and exploiting system scenarios to create more efficient embedded systems, describing generic solution strategies for the various methodology steps whenever possible.

3. SYSTEM SCENARIO METHODOLOGY

Although the concept of system scenarios has been applied before on top of concrete design techniques both in an ad-hoc [Chung et al. 2002; Hansson et al. 2007; Murali et al. 2006b; Sasanka et al. 2002] as well as in a systematic way [Gheorghita et al. 2005; 2008b; Hamers et al. 2007; Mamagkakis et al. 2007; Palkovic et al. 2006; Yang et al. 2002], it is possible to generalize all those scenario approaches into a common systematic methodology. This section describes such a general and still near-optimal methodology, providing generic solutions whenever available. This generic methodology is the most important contribution of this paper. Parts of the solutions provided for the various steps, in particular those presented in sections 3.4 and 3.7, have not been published earlier. The methodology is applied to some specific contexts in section 4. The section is structured as follows. In section 3.1, the basic concepts behind the system scenario methodology are described. The methodology overview is given in section 3.2. The remaining subsections refine each of the steps of the general methodology. In the subsequent subsections, we always refer to system scenarios also when we use the abbreviated term *scenario*.

3.1 Basic Concepts

The goal of a scenario method is, given a system, to exploit at design-time its possible RTSs, without getting into an explosion of detail. If the environment, the inputs and the hardware architecture status would always be the same, then it would be possible to optimally tune the system to that particular situation. However, since a lot of parameters are changing all the time, the system must be designed for the worst case situation. Still, it is possible to tune the system at run-time (e.g., change the processor frequency/supply voltage), based on the actual RTS. If this has to happen entirely during run-time, the overhead is most likely too large. So, an optimal configuration of the system is selected up front, at design-time. However, if a different configuration would be stored for every possible RTS, a huge database is required. Therefore, the RTSs similar from the resource usage perspective are clustered together into a single scenario, for which we store a tuned configuration for the worst case of all RTSs included in it.

The system scenario methodology deals with two main problems. First, scenarios introduce various overheads due to switching between scenarios, storing code for a set of scenarios instead of a single application instance, predicting the RTS, etc. The decision of what constitutes a scenario has to take into account all these overheads, which leads to a complicated problem. Therefore, we divide the scenario approach into steps. Second, using a scenario method, the system requires extra functionality: deciding which scenario to switch to (or not to switch), using the scenario to change the system configuration, and updating the scenario set with new information gathered at run-time.

Many system parameters exist that can be tuned at run-time while the system operates, in order to optimize the application behavior on the platform which it is mapped on. We call these parameters *system knobs*. A huge variety of system knobs is available. Section 2 gives the example of DVFS; entirely different examples of other possible system knobs include the version of the code to run in case of an application that contains multiple versions of its source code, different compiler optimizations being applied to each of them [Palkovic et al. 2006], and the configuration of processing elements (e.g., number and type of function units) in a multi-processor system [Sasanka et al. 2002]. Anything that can be changed about the system during operation and that affects system cost (directly or indirectly) can be considered a system knob. Note that these changes do not have to occur at the hardware level; they can occur at the software level as well. A particular position or tuning of a system knob is called a *knob position*. If the knob positions are fully fixed at design-time, then the system will always have the same fixed, worst case cost. By configuring knobs while the system is operating, the system cost can be affected. In the DVFS example, the knob position is the choice of a particular operating voltage, and its change directly affects the processor speed and power, and indirectly the energy consumed to execute the application. However, tuning knob positions at run-time introduces overhead, which should be taken into account when the system cost is computed.

Instead of choosing a single knob position at design-time, it is possible, and highly desirable, to design for several knob positions. At different occurrences during run-time, one of these knob positions is chosen, depending on the actual RTS. When

the RTS starts, the appropriate knob position should be set. Moreover, the knob position is not changed during the RTS execution. Therefore, it is necessary to determine which RTS is about to start. This prediction is based on *RTS parameters*, which have to be observable and which are assumed to remain sufficiently constant during the RTS execution. These parameters together with their values in a given RTS form the RTS *snapshot*. In the H.264 example, the RTS corresponds to the decoding of a frame, and the RTS parameter is the frame breakup into the macroblock types.

The number of distinguishable RTSs from a system is exponential in the number of observable parameters. Therefore, to avoid the complexity of handling all of them at run-time, several RTSs are clustered into a single *system scenario*. A tradeoff is present here between optimisation quality and run-time overhead of the scenario exploitation. At run-time, the RTS parameters are used to detect the current scenario rather than the current RTS. In principle, the same knob position is used for all the RTSs in a scenario, so they all have the same cost value: the worst case of all the RTSs in the scenario. Therefore, it is best to cluster RTSs which have nearby cost values. Since at run-time any RTS may be encountered, it is necessary to design not one scenario but rather a *scenario set*. A scenario set is a partitioning of all possible RTSs, i.e., each RTS must belong to exactly one scenario.

The approach presented above is clear when the cost is one-dimensional, i.e., when one cost aspect dominates or when all the different cost aspects have been combined in a normalized weighted sum. The latter is not always easy in practice because *"comparing apples and oranges"* in a single dimension usually leads to inconsistencies and suboptimal results. Hence, N-dimensional Pareto sets can be used to specify the costs of a system scenario consisting of different RTSs instead of a weighted one-dimensional cost. Such Pareto sets [Pareto 1906; Geilen et al. 2007] allow to work with a Pareto boundary between all feasible and all non-feasible points in the N-dimensional cost space. The Pareto boundary (the Pareto points) for all the possible RTSs that have been clustered into a scenario (and that can potentially be encountered at run-time) characterizes the scenario. Unfortunately, it becomes less obvious to deal with statements like *"nearby cost values of RTSs"* or *"taking the worst case of all the RTSs in the scenario"*. So, similarity between costs of different RTSs or in general sets of RTSs (scenarios) has to be substituted by a new element, e.g., by defining the normalized, potentially weighted distance between two N-dimensional Pareto sets as the size of an N-dimensional volume that is present in between these two sets. Based on this distance value, closeness of potential scenario options can be characterized, e.g., to decide whether or not to distinguish scenarios. An example of such an N-dimensional cost space and how to deal with it is provided in section 4.2. For more details, the reader is referred to [Okabe et al. 2003; Ykman-Couvreur et al. 2005; Ykman-Couvreur et al. 2006].

3.2 Methodology Overview

Even though the system scenario concept is applicable to many contexts, we have devised a general methodology that can be instantiated in all of these contexts. This system scenario methodology and the presented generic solutions for some of its steps deal with issues that are common: choosing a good scenario set, deciding

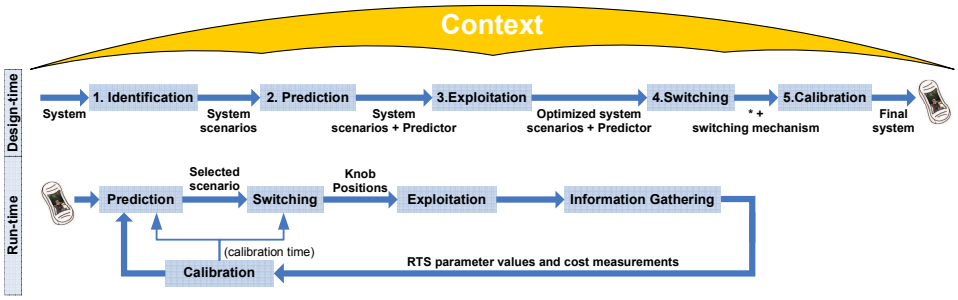

Fig. 5. The system scenario methodology overview.

which scenario to switch to (or not to switch), using the scenario to change the system knobs, and updating the scenario set based on new information gathered at run-time. This leads to a five step methodology (figure 5), each of the steps having a design-time and a run-time phase. The first step, identification, is somewhat special in this respect, in the sense that its run-time phase is merged into the run-time phase of the final step, calibration.

(1) *Identification* of the scenario set: In this step, the relevant RTS parameters are selected and the RTSs are clustered into scenarios. This clustering is based on the cost tradeoffs of the RTSs, or an estimate thereof. The identification step should take as much as possible into account the overhead costs introduced in the system by the following steps of the methodology. As this is not easy to achieve, an alternative solution is to refine the scenario identification (i.e., to further cluster RTSs) during these steps. Section 3.3 discusses the identification step in more detail.

(2) *Prediction* of the scenario: At run-time, a scenario has to be selected from the scenario set based on the actual parameter values. This selection process is referred to as scenario prediction. In the general case, the parameter values may not be known before the RTS starts, so they may have to be estimated. Prediction is not a trivial task: both the number of parameters and the number of scenarios may be considerable, so a simple lookup in a list of scenarios may not be feasible. The prediction incurs a certain run-time overhead, which depends on the chosen scenario set. Therefore, the scenario set may be refined based on the prediction overhead. Section 3.4 details the two decisions made by this step at design-time: selection of the run-time prediction algorithm and refinement of the scenario set.

(3) *Exploitation* of the scenario set: At design-time, the exploitation step is essentially based on some optimization that is applied when no scenario approach is applied. A scenario approach can simply be put on top of this optimization by applying the optimization to each scenario of the scenario set separately. Using the additional scenario information enables better optimization. At run-time, the exploitation is in fact the execution of the scenario. Section 3.5 details the common problems that should be handled during the exploitation step.

(4) *Switching* from one scenario to another: Switching is the act of changing the system from one set of knob positions to another. This implies some overhead

(e.g., time and energy), which may be large (e.g., when migrating a task from one processor to another). Therefore, even when a certain scenario (different from the current one) is predicted, it is not always a good idea to switch to it, because the overhead may be larger than the gain. The switching step, detailed in section 3.6, selects at design-time an algorithm that is used at run-time to decide whether to switch or not. It also introduces into the application the way how to change the knob positions, i.e., how to implement the switch, and refines the scenario set by taking into account switching overhead.

(5) *Calibration*: The previous steps of our methodology make different choices (e.g., scenario set, prediction algorithm) at design-time that depend very much on the values that the RTS parameters typically have at run-time: it makes no sense to support a certain scenario if in practice it (almost) never occurs. To determine the typical values for the parameters, profiling augmented with static analysis can be used. However, our ability to predict the actual run-time environment, including the input data, is obviously limited. Therefore, we also foresee support for infrequent calibration at run-time, which complements all the methodology steps previously described. At design-time, information gathering mechanisms are designed and added to the application. At run-time they collect information about actual values of the parameters and the quality of the resulting system (e.g., number of deadline misses). Besides this, a calibration mechanism is introduced in the application. This is used to calibrate the cost estimates, the set of scenarios, the values of the parameters used for scenario prediction, and the knob positions. Calibration of the scenario set does not take place continuously during run-time, but only sporadically, at *calibration time*. Otherwise the overhead would obviously become too large. Also note that calibration is not meaningful when quality constraints are hard. It can only be applied if constraints are soft, or to optimize average-case behavior in the absence of constraints (e.g., when optimizing memory usage for energy reduction). Section 3.7 presents techniques for calibration.

In the following two paragraphs, we indicate intuitively why, in the design-time part of the methodology, the steps have been ordered as proposed. In particular, the reasoning behind this is based on a gradual pruning of the possible final scenario decisions. First, during identification, RTS parameters are limited to the ones that have a sufficient and observable cost impact on the final system. Then during clustering, we select the parameters that are easiest to be controlled as the actual system knobs and we cluster the corresponding RTSs based on similarity of cost (and knob settings, if applicable). In this way, we ensure that the cost distance between any two scenarios is maximized. This is needed because we have a clear tradeoff between the gains by introducing more scenarios and the cost it involves. This tradeoff leads to a further pruning of the search space for the most effective final scenario decisions. In the prediction step, we have to limit the potentially most usable scenarios to the ones that are also predictable at run-time with an affordable overhead. Also here a global tradeoff between gain and cost (run-time prediction overhead) is present. We cannot perform this second step of our method prior to the identification step because we cannot estimate the prediction cost before we at least have a good idea about the clustering of RTSs in scenarios. Note that the

opposite is not true: the information of the prediction step is not essential to decide on the clustering. This creates an asymmetrical relation which is the basis for the split between the two steps (see also the constrained orthogonalization approach in [Catthoor 2000]).

Only when we have decided how to perform the prediction, we can start the exploitation of the resulting scenarios in the particular application domain (step 3). Indeed, we could already start the exploitation after having the first clustering step, but that is not always efficient: the knowledge of the prediction cost will give us more potential for making good exploitation decisions. In contrast, the knowledge of the exploitation itself is not yet needed to make a good pruning choice on the prediction related selection. Finally, in the proposed design trajectory, we only decide on the scenario switching based on the actual overhead that is involved in the switching. The latter is only known after we have decided how to exploit the scenarios. The calibration step can be applied only when the rest of the steps are already done, as information about the scenario set, and the prediction and switching algorithms are needed to design the information gathering and calibration mechanism. So every step of our design-time methodology is positioned at a location where it has maximal impact but also where the required information to effectively decide on it is available as much as possible. The proposed split up in steps and order avoids phase coupling to a large extent. This avoids iteration on any of the individual steps after completion of a subsequent step in the methodology, which is a deliberate and important property of our generic design methodology.

The ordering of the steps at run-time follows the natural ordering of the various activities as they are needed at run-time. The ordering is in line with the design-time ordering with two exceptions. The first one was already mentioned. The identification and calibration steps are integrated, because part of the run-time calibration step may be to identify new scenarios, and no other means to identify new scenarios at run-time are available. Furthermore, the order of switching and exploitation is reversed when compared to the design-time order, as the run-time switching prepares the system for the given exploitation.

The next subsections detail the various steps of the methodology, outlining generic solutions whenever such solutions are available. The implementation of predictors presented in section 3.4 and the calibration mechanisms of section 3.7 have not been published earlier. All the presented solutions have been fully automated, and are applied in the various case studies presented in section 4.

3.3 Identification

Before gaining the advantages that a scenario approach gives, it is necessary to identify the different scenarios that group all possible RTSs. This identification process happens in two phases, RTS parameter discovery and RTS clustering.

First, the interesting snapshot parameters are discovered. As mentioned before, a snapshot contains all parameters as well as their values that characterize a certain RTS. However, we are only interested in those parameters which have an impact on the system's behavior and execution cost. For example, interesting RTS parameters for an audio-video system are the size of the video frame, and whether the audio stream is mono or stereo.

The values of the selected parameters will be used to distinguish between the different RTSs, so two RTSs with the same snapshot are considered identical. However, they may still have different actual cost values, due to a choice of the parameters that does not precisely capture all the unique system behaviors. For example, two RTSs with a different data-dependent loop bound have a different execution time, but we consider them the same RTS if we are not observing that loop bound. When we are also observing that loop bound, each number of iterations corresponds to a different RTS. In the general case, a parameter selection that does not precisely capture all the individual behaviors of a system may lead to RTSs with a set of Pareto points in the multi-dimensional cost space as their actual (worst case) cost values. Such a parameter selection may be due to an imperfect analysis or for complexity reasons. However, it may also be deliberate, e.g., with the intention to handle certain minor dynamic variations or configuration options (i.e., a choice among different knob positions) entirely at run-time, or to create a hierarchy of scenarios, where the variation within one scenario at a certain hierarchical level is handled by another set of (sub-)scenarios.

Second, following the parameter discovery, all possible RTSs are clustered into system scenarios based upon a multi-objective cost function. The multi-objective cost function is dependent on the specific optimization and the system knobs we have in mind for the exploitation step. For the H.264 decoder presented in section 2, which targets a single-processor platform, we aim at reducing energy by applying DVFS and so we need accurate cycle-budget estimations for decoding the frames. The objective function in this case is one-dimensional and it is represented by the cycle budget needed for decoding each frame. (Note that the decoding of a frame was considered the RTS in this example.) For multi-media applications running on a multi-processor platform, the cost is typically multi-dimensional, including for example cycle budgets per processor, the derived frame decoding times and, when aiming at power optimization, derived power budgets. Whereas the knob for configuring a single-processor system for power optimization via DFVS is simply a single frequency setting, the knobs in a multi-processor system may potentially include configuration options of the various processors, the interconnect, the memory hierarchy, and other platform components. The RTS clustering into scenarios needs to take both cost and knob settings into account, to guarantee that only RTSs with compatible knob settings and similar costs are clustered into a single scenario.

The remaining part of this section details the two phases of the identification process.

3.3.1 *RTS Parameter Discovery.* In related work done so far, usually, parameter discovery is done in an ad-hoc manual manner by the system designer, by analyzing the application and profiting from domain knowledge. This is fine when all the important parameters are immediately obvious, such as the frame size in a video decoder. However, this process might prove tedious and incomplete for complex systems, as parameters that may have a large impact on the system behavior might go unnoticed. Developing a fully general tool that discovers the interesting parameters for all the design approaches where scenarios may be applied is hard, maybe even impossible, to realize due to the diversity of cost functions and optimization objectives. Therefore, we have developed a quite broadly applicable

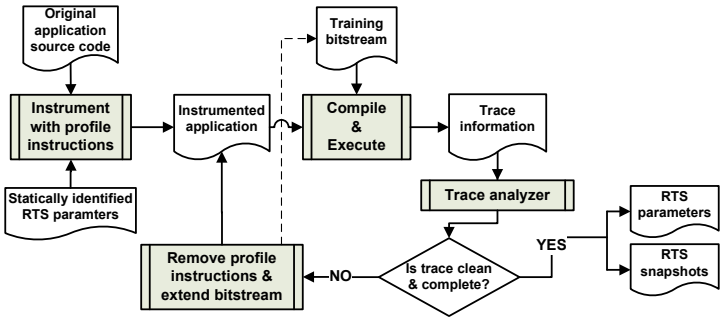

Fig. 6. RTS parameter identification approach.

domain-specific approach that is presented in the remaining part of this subsection.

Our tool searches for control variables in the application source code that have a certain impact on the application resource requirements (e.g., number of cycles, memory utilization). These parameters fulfill the two requirements for selection: they are observable and they influence the application's behavior and cost (i.e., the resource needs). A first version of this tool [Gheorghita et al. 2005] statically analyzes the application source code to identify these variables. It is applicable for hard (real-time) constraints, due to the conservative analysis. In [Gheorghita et al. 2008b], a version applicable for soft real-time systems is presented. It profiles the application, and it uses the collected information for eliminating those control variables whose values do not have a real impact on system cost.

Our profiling based approach, detailed in figure 6, starts from the application source code which is then instrumented with profile instructions for all read and write operations on the statically identified variables. The instrumented code is then compiled and executed on (the initial part of) a training bitstream and the resulting program trace is collected and analyzed. The trace analyzer identifies the variables which do not have a large influence on the application behavior using different heuristics (e.g., it identifies the instructions that read and write to a large number of variables, which resembles an array-like access pattern; the accessed variables are considered to be data variables which have a small influence on the system behavior). In the next iteration, the instrumentation for these variables is removed and the application is executed on a larger part of the training sequence. This procedure is repeated until the entire sequence is processed and the trace analyzer does not dismiss any more variables. In each step of the design, and at each iteration, manual intervention is possible, but not necessary, to steer the decision process. The final result of this automated discovery is thus a list of relevant RTS parameters. During the profiling step it is of course possible to collect additional information such as the met RTSs identified by their snapshot, together with their cost. This information is then used in the RTS clustering step. However, finding a *representative* training bitstream that covers most of the behaviors that may appear during the system life-time, particularly including the most frequent ones, is in general a difficult problem. Hence, in contrast with analysis based identification that covers all possible RTSs, the profiling based identification is not conservative. It can happen that, at run-time, when the system runs, an RTS

that was not considered during identification is met. Therefore, a way of handling this situation should be added in the final implementation of the system. The calibration step in our method (see section 3.7) has been included for this reason, among others. Experiments show that the combination of profiling based parameter discovery and calibration is quite robust (see section 4.1), alleviating the problem of finding representative training sets and reducing the time needed for training.

3.3.2 *RTS Clustering.* Using the discovered RTS parameters, all identified RTSs are clustered into a set of system scenarios. For each RTS, in the most general case, we have a Pareto surface of potential exploitation points in the multi-dimensional exploration space. The clustering is done based upon a multi-objective cost function which is related to the specific optimization we want to apply to the system. Hence, the clustering searches for RTSs with similar Pareto surfaces. It starts from RTS snapshots and generates a set of scenarios, each of the scenarios being identified by a set of snapshots. The clustering takes into account the following information: (i) how often and for how long each RTS occurs at run-time, (ii) the cost deviation that occurs when clustering multiple RTSs into a single scenario, (iii) how many switches occur between each pair of scenarios, and (iv) the run-time scenario prediction, storage and switching overhead. It furthermore has to make sure that knob settings for the RTSs clustered in a single scenario are compatible. Having a multi-dimensional cost function means that both the inherent scenario costs and the switching cost becomes multi-dimensional also. Creating a good scenario set under these constraints can be formulated as an N-dimensional optimization problem. However, this optimization problem does not have a general practically executable solution, so heuristics need to be developed.

When clustering different RTSs into a scenario, we determine the cost of the scenario as the maximal cost of the RTSs that compose the scenario (which in a multi-dimensional cost space results in a Pareto surface). The clustering process is driven by two opposing forces. One force drives towards a large number of scenarios that contain a few RTSs, the extreme being each scenario to contain only one RTS, by only grouping RTSs with similar cost together, so that the estimated deviation between the cost value of an RTS and the cost of the scenario remains small. It uses the information from items (i) and (ii) of the list above. The other force takes into account the overheads (items (iii) and (iv) of the above list) introduced by the existence of a large number of scenarios, and it aims to decrease the number of scenarios by increasing their size in the number of contained RTSs.

Since the application does not remain in the same scenario forever, the switching overhead has to be taken into account. This overhead usually has effects on the cost function (e.g., scaling frequency and voltage of the processor costs both time and energy). So, depending on how large the switching overhead is, the aim is to reduce the number of scenario switches that appear at run-time. Taking this into account, the two forces identified above have to generate a tradeoff by clustering together into a scenario, not only RTSs with similar cost, but also the ones between which many switches appear at run-time.

The storage overhead of scenarios is strongly dependent on the kind of optimizations that are applied in the exploitation step. For example, in the H.264 decoder presented in section 2, a table has to be kept which maps the different scenarios

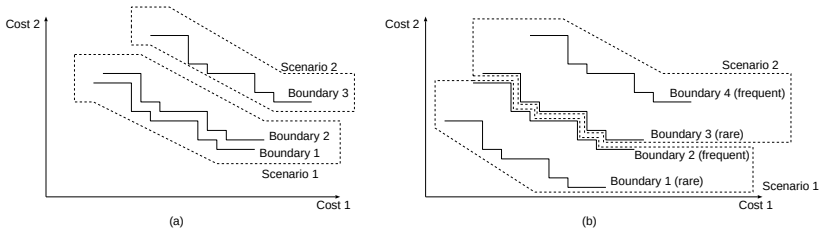

Fig. 7. Examples of RTS clustering.

to the optimal frequency-voltage pair. When the number of scenarios increases so does the size of this table, but the overhead per scenario will be small. On the other hand, in the case study presented in section 4.2, when optimized code is generated for each separate scenario, the overhead for storing this scenario-specific code is rather large.

Finally, since the scenarios need to be predicted at run-time, there is also the scenario predictor to consider. If the amount of scenarios increases, it will result in a larger and perhaps slower predictor. Also, the probability of a faulty prediction may increase with the number of possible scenarios.

Figure 7(a) depicts a clustering example with a two-dimensional cost function. Each RTS is represented by a (two-dimensional) Pareto boundary which represents different RTS knob configurations. Usually, the RTSs with similar Pareto boundaries are clustered to one scenario at design-time. Thus, the distance between two Pareto boundaries determines which ones to cluster. The cluster is then represented by its worst-case Pareto boundary. Based on this criterion, boundaries 1 and 2 are clustered in figure 7(a), forming the first scenario. Boundary 3 forms alone the second scenario. As already mentioned, apart from the distance also the frequency of occurrence is important. If a very frequent RTS is clustered with a very rare one, which has a worse Pareto boundary, this scenario and all RTSs in this scenario inherit the worse, rare Pareto boundary. In such a case, it is better to cluster the frequent Pareto boundaries with better rare Pareto boundaries so that the frequent Pareto boundary represents the created scenario. This is depicted in figure 7(b). This analysis and clustering is done at design-time. At run-time, the appropriate scenario with its Pareto boundary is identified and a concrete Pareto point is selected related to a specific knob configuration.

3.4 Prediction

This step aims at deriving a predictor, which can determine at run-time the appropriate scenario in which the system executes. It starts from the information collected in the identification step. Just as this parameter identification step, scenario prediction can be solved in a rather generic way that is widely applicable. The resulting predictor mainly bases its decision on the values of the RTS parameters. Moreover, it has to be flexible (e.g., to have a structure that can be easily modified during the calibration phase) and to add a small decision overhead in the final system. We can define it as a prediction function:

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

where n is the number of RTS parameters, Ω_k is the set of all possible values of the parameter ξ_k (including $\sim$ that represents undefined) and m is the number of scenarios in which the system was divided. The function f maps each RTS i, based on the parameter values $\xi_k(i)$ associated with it, to the scenario to which the RTS belongs. If at run-time an RTS occurs that was not considered during the identification phase (e.g., because it was not met during profiling), it is mapped to the scenario that can deal even with the worst-case situation, the so-called *backup scenario*. If any optimization takes place, it is based on a worst-case analysis at design-time.

A predictor based only on the prediction function approach can be applied only after all the parameter values are known. If the identification was done in a conservative mode, which covers all possible RTSs that may appear at run-time, the prediction accuracy will be 100%, and we can speak about scenario *detection*. Waiting until all the parameter values are known at run-time may postpone the prediction moment unnecessarily long, and the scenario may be predicted too late to still profit maximally from the applied optimization. To handle this problem, multiple approaches may be considered (not necessarily in isolation), like (i) reducing or changing the set of considered parameters, and (ii) combining the prediction function with pure probabilistic prediction. In the first approach, we search for the set of parameters that can be used to identify the set of predictable scenarios that gives the highest gain, taking into the account the moment when they can be predicted at run-time. In the second case, the scenario prediction point may be moved to an earlier point in time by augmenting the prediction function with a mechanism that selects from the possible set of scenarios predicted by the function, the one with highest probability. For example, the mechanism may use advanced phase predictors [Vandeputte et al. 2005]. Using the probabilistic approach, the miss-prediction may increase. It is of two types: (i) over-prediction, when a scenario with a larger (multi-dimensional) cost is selected, and (ii) under-prediction, when a scenario with smaller (multi-dimensional) cost is selected. The first type does not produce critical effects, just leading to a less cost effective system; the second type often reduces the system quality, e.g., by increasing the number of deadline misses for the H.264 decoder.

The place where the prediction function is introduced into the application, is called a *scenario prediction point*. From a structural point of view, considering the number of times and the places where the prediction function is introduced into the application, the predictors can be classified as follows:

—*Centralized*: There is only one central point in the application where the current scenario is predicted. It is inserted in the application code in a common place that appears in all scenarios. For example, in the case of the application model presented in figure 8(a), it is introduced in the main loop, after the read part, when all the information necessary to predict the current scenario is known.

—*Distributed*: There are multiple scenario prediction points, which may be:
  —*Exclusive points*: An identical (or tuned) prediction function is introduced multiple times into the application, in all the places where the RTS parameter values are known. At run-time, only one point from the set is executed in each RTS. This kind of predictor solves the problem that there may be no

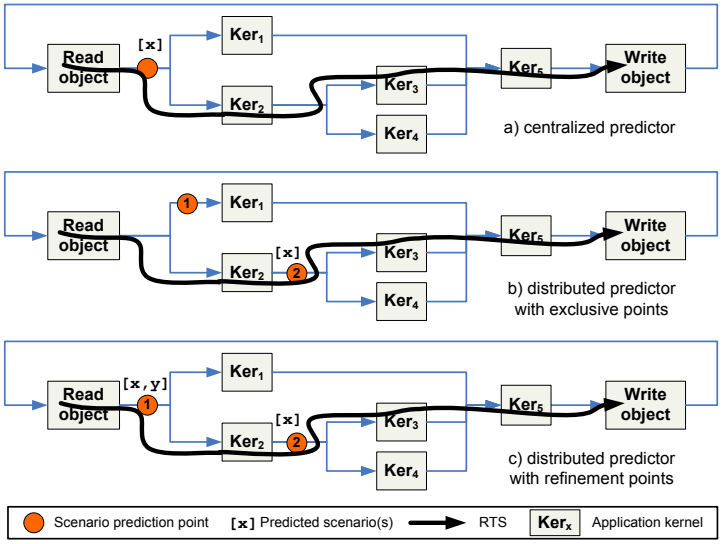

Fig. 8. Types of scenario prediction.

common place in all scenarios, where a centralized predictor may be inserted. Figure 8(b) depicts a case where one of two prediction points is being executed for different RTSs.

—*Refinement points*: Multiple points, which work as a hierarchy, are used to predict the current scenario in a loop iteration; the first that is met at run-time predicts a set of possible scenarios, and the following refine the set until only one scenario remains. This extension can improve the efficiency of optimizations as earlier switching between scenarios may be done, but it increases the number of switches. Hence, a tradeoff should be considered when using it, which depends on the problem at hand. This is actually a very similar problem tradeoff as the one for the RTS clustering substep. Hence, we believe that also here a similar set of heuristic algorithms can be reused. Usually, when switching between scenarios after a refinement predictor, the new scenario may be the scenario with the worst case cost from the remaining set of scenarios. However, the probabilistic approach presented above could also be used to select the scenario to which to switch. For the example depicted in figure 8(c), considering the scenario that executes kernels two, three and five, in the first scenario prediction point the set containing scenarios x and y is selected. Then, in the second scenario prediction point, the set is refined to only one scenario, x.

A generic implementation of a prediction function f, which incorporates requirements like flexibility and small overhead, is a *multi-valued decision diagram* [Wegener 2000], which 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

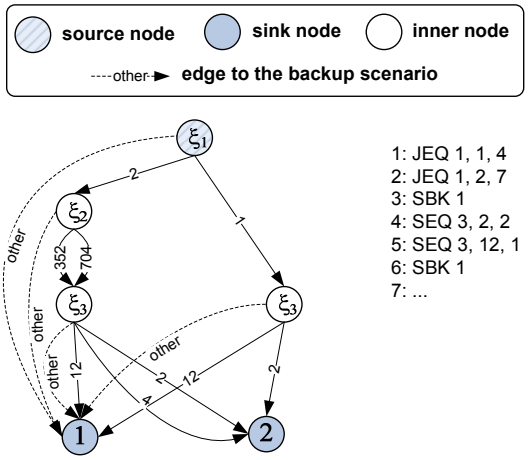

Fig. 9. An example of a decision diagram and its implementation.

OP	VARIABLE-ID	VALUE	DATA	Description
JEQ	<var>	<val>	<address>	Jump to <address> if <var> is equal to <val>
JL	<var>	<val>	<address>	Jump to <address> if <var> is less than <val>
JMP	-	-	<address>	Unconditional jump to <address>
SEQ	<var>	<val>	<scenario>	Predict <scenario> if <var> is equal to <val>
SLE	<var>	<val>	<scenario>	Predict <scenario> if <var> is less or equal to <val>
SBK	-	-	<scenario>	Predict <scenario> as a backup scenario

Table I. Instruction set used in predictor implementation.

RTS parameter ξ_k during the identification phase plus an edge labeled with *other* that leads directly to the backup scenario. 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 RTS parameter ξ_k occurs at most once. A simplified example of a decision diagram for the motion compensation kernel from the H.264 video decoder is shown in figure 9. When a prediction function is used, it introduces two overheads: (i) *code size* and (ii) *average run-time evaluation cost*. Also this is similar to the RTS clustering substep. In the proposed solution, both depend on the decision diagram size (number of nodes and edges). Hence, reducing its size, the overheads decrease. However, the applied reduction rules may affect the prediction quality, especially for the RTSs that were not considered during the identification step (because for these RTSs, a different scenario may be predicted instead of the backup scenario). A few reduction rules are analyzed in [Gheorghita et al. 2008b]. Note that these reductions may render certain RTS parameters redundant, which happens if they do no longer occur in the final predictor. Also note that, conceptually, decision diagram transformations are scenario refinements, because a transformation affects the parameter value ranges characterizing a scenario.

We propose to implement decision diagrams as a program in a restricted programming language. The language is specified in table I and an example of its use is given in figure 9. The program implementing a decision diagram is represented by

```
predictScenario(HashTable values,Vector dd)
1  pc ← 1
2  while TRUE
3        do value ← values[dd[pc].variable-id]
4           if (dd[pc].op = jeq and value = dd[pc].value) or
                 (dd[pc].op = jl and value < dd[pc].value) or (dd[pc].op = jmp)
5             then pc ← dd[pc].data
6           elseif (dd[pc].op = seq and value = dd[pc].value) or
                      (dd[pc].op = sle and value ≤ dd[pc].value) or (dd[pc].op = sbk)
7             then return dd[pc].data
8             else pc++
```

Fig. 10. Decision diagram execution engine.

a data array in the application source code and it is executed by a simple execution engine, given in figure 10 and explained below. The proposed implementation has very little overhead and it allows an easy calibration of the decision diagram, by changing the values of appropriate array elements.

The language defined in table I allows to implement each edge of a decision diagram by one instruction, by using (i) a `JEQ` instruction if its destination node is labeled with a variable name, or (ii) a `SEQ` instruction if it is not labeled `other` and its destination node is labeled with a scenario name, or (iii) a `SBK` instruction if it is an `other` labeled edge. To reduce implementation costs, a group of adjacent edges that have a common destination can be implemented using only two instructions, using the `JL` and `SLE` instructions. The `JMP` instruction is used for calibration, as explained below.

The program that represents the decision diagram is executed by the execution engine presented in figure 10. This engine receives as input parameters a hash table (*values*) containing the pairs variable/value for the current RTS, and a vector (*dd*) containing the program that has to be executed. Each vector element represents an instruction. The position of the instruction to be executed is kept in the program counter *pc*, which is initialized to start with the first program instruction (line 1). The program execution ends only when an instruction that sets a scenario is executed and its condition, if present, evaluates to true (lines 6-7). If a jump instruction is met and its condition evaluates to true, the next instruction to be executed is determined by the data field of the current jump instruction (lines 4-5). Otherwise, if no condition evaluates to true, the program counter is set such that the next sequential instruction will be executed (line 8).

In conclusion, the above has detailed a generic form and implementation of scenario predictors, that is flexible and well suited for calibration, as explained in more detail in section 3.7. We end this section by mentioning several other aspects that may be addressed during the prediction step at design-time. The prediction step may cover the following actions: (i) a further clustering of scenarios considering the prediction overhead and the moment when the scenario may be predicted, (ii) possibly, a further pruning of the RTS parameters, (iii) clustering of previously unassigned RTSs (i.e., the ones that were not met during the identification process) into scenarios, and (iv) defining and placing the prediction mechanism into the application, by trading off prediction accuracy versus overhead, which influence the final system cost and quality.

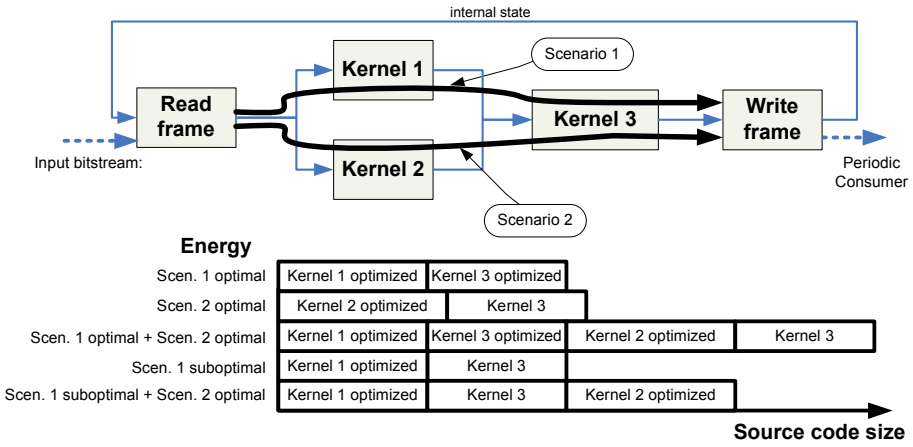

Fig. 11. Scenario source code merging.

3.5 Exploitation

The exploitation step in the scenario methodology is very dependent on the context in which scenarios are applied. Nevertheless, some general aspects to be kept in mind can be mentioned. Exploitation in the context of the scenario methodology should be refined in two ways, to a large extent independently of the type of exploitation. First, optimizing each scenario in isolation might be inefficient. There is a strong correlation between the analysis and the optimization choices of the different scenarios, so the optimization of a scenario can be performed more efficiently by reusing information from other scenarios. Second, separate optimization for each scenario leads to separate systems. Simply putting all these next to each other could imply a huge overhead. Therefore, whatever is common between different scenarios should be merged together, e.g., by using code compaction techniques [Debray et al. 2002; De Sutter et al. 2006]. The remaining differences cause exploitation overhead, which should be taken into account to further refine the scenario set. Some optimizations that are suboptimal for an individual scenario, might be optimal from the system cost perspective when considering exploitation overhead. How difficult it is to simultaneously optimize scenarios depends on the context. As an example, figure 11 depicts an application with two scenarios: scenario 1 for the case where kernels 1 and 3 are executed, and scenario 2 for the case where kernels 2 and 3 are executed. To optimize the application for energy, a compiler may optimize each scenario separately to reduce the number of computation cycles. Assume that the energy-optimal exploitation of each scenario is, for scenario 1, to optimize both kernels 1 and 3, and, for scenario 2, to optimize only kernel 2, kernel 3 already being energy optimal for this scenario. Combining these two optimal scenario exploitations, the application source code contains code for kernel 3 twice (once optimized for scenario 1, and once untouched, as used in scenario 2). If the energy overhead introduced by storing the two versions of kernel 3 is large, the energy-optimal system might be obtained by using a suboptimal version of scenario 1, as presented in figure 11. This version uses the original implementation of kernel

3, so no code duplication for this kernel is needed in the final implementation of the application.

Both mentioned exploitation refinements for scenarios are specific to the type of optimization that is performed, so exploitation cannot really be fully generalized, illustrative examples being given in the literature overview of section 5 and the case studies of section 4.

3.6 Switching

A system execution is a sequence of RTSs, and therefore a sequence of scenarios. At the border between two scenarios during execution, switching occurs. For executing this switch at run-time, at design-time a mechanism is derived and introduced into the system. The switching decision and process (changing the knob positions) may incur overhead, which is taken into account to further refine the scenario set. Moreover, it is also taken into account at run-time to decide whether or not to switch to a different scenario, together with other information (i.e., the sequence of previous and possible following RTSs). The expected gain times the expected time window where the scenario is applicable has to be compared to the exploitation cost, as already mentioned. The structure of this switching mechanism should be flexible enough to allow it to be calibrated.

Even if the switching overhead is exploitation dependent, our methodology treats this overhead in a general way. It uses the scenario cost versus overhead reports (e.g., energy, time) together with the information about how often a switch between two given scenarios appears at run-time, to avoid spending most of the system's running time on switching between scenarios, instead of on doing relevant work. For our H.264 example from section 2, the switching operation adjusts the supply voltage and processor frequency. Its overhead in time and energy depends on the implementation. Using the hardware circuit presented in [Burd et al. 2000] for switching, the overhead measured in time is up to $70\mu s$ and in energy up to $4\mu J$. These overheads affect both the final system cost (e.g., more energy consumption) and its run-time properties (e.g., more deadline misses because of time overhead). It is important to compare the time overhead with the minimum time the system stays in a scenario, which is equal to the required period between two consecutive frames (or smaller due to late scenario prediction). For a throughput of 30 frames per second, a switch may be acceptable between each pair of consecutive frames, as the overhead represents up to 0.2% of the time ($70\mu s$ out of $33ms$). On the other hand, for an application with for example 2500 RTSs per second, the switch overhead per frame represents 20% of the time, so the switches should be quite rare. Another illustration of the very small overhead of the run-time controller is reported in [Yang and Catthoor 2003].

The way how the exploitation step encodes the scenarios into the system affects the switching cost. As we already mentioned, in the H.264 example, for each scenario a frequency-voltage pair is stored. However, for other exploitation examples, like the one presented in section 4.2, a copy of the source code for each scenario should be stored. These copies introduce large supplementary costs into the final system for each added scenario, and limit the total number of scenarios. The technique presented in section 4.2 can find a scenario set which achieves the best data memory optimization for the given instruction memory overhead. For a scenario

that is rarely activated, its source code may be kept in a compressed version to reduce the storage cost, but as a decompression is done when the scenario is enabled, this increases the switching overhead.

Thus, the overhead for switching between two scenarios depends on what the run-time switching implies, and the scenarios between which the application switches. The switching overhead affects both the final system cost (e.g., more energy consumption) and its run-time properties (e.g., more deadline misses because of time overhead). In the switching step at design-time, in parallel with deriving the switching mechanism, the set of scenarios, and consequently the predictor, may need to be adapted. This adaptation takes into account the cost of each scenario, how often the switch between each pair of scenarios appears at run-time and how expensive it is. Two scenarios that were considered separately so far but which have a relatively close cost, and between which the system switches very often at run-time might be merged in a scenario with the worst case cost among them.

The time overhead introduced by switching may cause undesired side-effects in a system, such as deadline misses in an H.264 decoder. Besides system and context dependent ways of handling such side-effects (e.g., an H.264 decoder may display the previous frame again if the deadline for the current frame is missed), we looked at a general way for minimizing the side-effects that are caused by the time overhead introduced by the switching mechanism. The most conservative way to handle this overhead is to reserve time in each scenario, considering that the scenario is always activated for only one RTS in a row and taking into account the largest switching time that may appear. This approach might be very expensive, in which case it is a viable solution only for systems that require hard guarantees. For systems where more freedom is acceptable, in each scenario, we may reserve time considering the switching time overhead averaged over the typical number of subsequent RTSs spent in a scenario, and the possible over-estimation in timing requirements that exist in the scenario. Such an over-estimation appears because for all RTSs clustered into a scenario, their worst case cost is considered always when the scenario appears. Moreover, buffers exist in almost all modern systems, such as an output buffer in a video decoding system, which potentially can be used to compensate for the overhead variations that appear at run-time.

3.7 Calibration

The previous presented steps of our methodology make different design-time choices (e.g., scenario set, prediction algorithm) that depend very much on the possible values of RTS parameters, typically derived using profiling. This approach is obviously limited by our ability to predict the actual run-time environment, including the input data. It may lead to run-time problems, like encountering an RTS that was not considered in the design-time choices, or an RTS with a higher cost than the one of the scenario to which it is predicted to belong. The first case appears when an RTS occurs at run-time of which the snapshot was not met during the identification step. In the second case, its snapshot was considered during the identification step, but the worst case cost observed for that snapshot is smaller than the actual cost of this RTS. This is also related to a possibly imperfect choice of the parameters or simplification of the predictor. Therefore, calibration can be used at run-time to complement the methodology steps previously presented. And even if RTS pa-

rameters guarantee correct prediction and costs are conservative, calibration can be useful to exploit for example a situation in which low cost RTSs occur for long periods of time in a row.

At run-time, information is collected about actual values of the RTS parameters, the predicted scenario, the decisions taken by the switching mechanism, the measured cost for each scenario prediction and the quality of the resulting system (e.g., the number of deadline misses). Both the execution cycles of the collecting process and the amount of stored information should be small as the collection is executed for each RTS. To keep the overhead limited, the calibration mechanism therefore has access to only a limited amount of information. Moreover, it should be implemented as a low complexity algorithm.

Periodically, sporadically (e.g., when time slack is found into the system), or in critical situations (e.g., when the system quality is too low due to a certain number of missed deadlines), the calibration mechanism is enabled. Based on the collected information, it may (i) change the ranges of parameter values and knob positions that characterize each scenario, and (ii) adapt the scenario set by clustering existing scenarios or introducing new ones. In these cases, the prediction, and maybe the switching mechanism have to be adapted as well. However, during the calibration, no new parameters or knobs are added, because this leads to a complicated and expensive process, as to exploit the new parameters the predictor should be redesigned and for the new knobs the scenario exploitation step should be redone.

Depending on the optimization applied in the exploitation step, the most common operations in the two above mentioned categories that can be done efficiently considering the calibration's limited processing and storage budgets are:

(1) To consider new RTSs that were not considered at design-time, and to map them to the scenario where they fit the best, based on the cost function, or to a new scenario. In this case, the predictor and the switching mechanism are also extended. As the complexity of the extension algorithm should be low, the resulting predictor will in general not be as efficient as if a new predictor were derived from scratch taking into account these new RTSs. Moreover, because an explosion in scenario storage has to be avoided, not for each RTS a new scenario can be created, but only for the ones which appear frequently enough to be promising for our final objective or problematic in terms of system quality.

(2) To increase the actual cost of a scenario, based on its RTSs observed at run-time. This case may appear because the RTSs are defined using a limited set of parameters, and it is possible that there exist multiple equivalent RTSs with different cost and only the cheaper ones were considered at design time. The same problem may occur also when prediction quality is low, if many RTSs are incorrectly predicted to belong to a scenario with a cost that is too low (under-prediction).

(3) To increase the cost of some or all scenarios, because the run-time overhead introduced by related scenario mechanisms (e.g., prediction) is higher than anticipated. The same problem appears when the run-time overhead variations are too high and the buffering in the system can not handle those variations. These cases are related to the fact that the input data and the environment in which the system runs is an extreme case (e.g., a lot of scenario switches), and

```
CALIBRATION(INT RTSCounter, ...)
 1  INFORMATIONGATHERING()
 2  SMALLADAPTATIONS()
 3  for i ← 1 to noCriticalCalibrations
 4       do if (RTSCounter − cCalib[i].LASTACTIVATION > cCalib[i].PERIOD)
 5            then cCalib[i].FN(...)
 6                 cCalib[i].LASTACTIVATION ← RTSCounter
 7  for i ← 1 to noNonCriticalCalibrations
 8       do if (RTSCounter − nCalib[i].LASTACTIVATION > nCalib[i].PERIOD)
 9            then if ENOUGHSLACK(nCalib[i].WCEC)
10                   then nCalib[i].FN(...)
11                        nCalib[i].LASTACTIVATION ← RTSCounter
```

Fig. 12. Calibration structure.

the system was dimensioned for the average case.

(4) To decrease the cost of a scenario, when only the RTSs with a low cost from that scenario actually appear at run-time. This improves our system cost (e.g., reducing energy), but potentially affects system quality negatively when RTSs with a higher cost occur. To maintain system quality, the cost may be increased again via the mechanism described in item two of this list, or by monitoring the scenario, the scenario cost may be reset to the value that it had before calibration when one or a few of its RTSs with a higher measured cost than the current scenario cost occur.

As an example of the introduction of a new scenario, consider the predictor decision diagram and its implementation given in figure 9. Assume a new scenario 3 is defined for the case that variable ξ_1 has value 7. This can be incorporated in the implementation of the diagram by changing line 3 into a `jmp x` instruction to jump to some given line `x` corresponding to an empty entry in the array implementing the decision diagram. Line `x` is then set to `seq 1, 7, 3` and line `x+1` to `SBK 1`. In this way, every newly introduced scenario corresponds to two lines in the program implementing the decision diagram. This is not the most concise representation, but it is very simple to implement, very simple to revert, if necessary, and it allows to reserve space in the decision diagram implementation for a fixed number of new scenarios to be added at run-time.

All the operations presented above have the role to control and to guarantee the system quality, and to further improve our objective (i.e., to reduce the system cost) by exploiting the information collected at run-time. Our implementation inserts in the final application some calibration code in line with the structure presented in figure 12. This code is executed immediately after each RTS was executed. While the information gathering (line 1) and the small adaptations (line 2) are executed for each RTS, the different calibration algorithms are executed periodically (lines 3-11) to limit the introduced overhead and to give a chance to the system to become stable between two consecutive calibrations. The small adaptations are low complexity algorithms which are enabled usually when (i) severe quality problems occur, and the adaptation cannot be delayed as the problems will really bother the end user, or (ii) collecting and storing the information for a later calibration is more expensive than executing the calibration on the spot. Moreover, these adaptation algorithms usually update the currently selected scenario, while the calibration algorithms

examine and calibrate all possible scenarios of the system.

To avoid introducing too much processing overhead in the processing of one RTS, each calibration algorithm has a different activation period. Moreover, the algorithms are divided in two categories: (i) critical algorithms (lines 3-6) and (ii) non-critical algorithms (lines 7-11). The critical ones usually deal with the application constraints (e.g., deadlines or image quality), and are executed with an exact period. The non-critical ones deal with runtime tuning for cost reduction, and they can be postponed until enough slack remains after processing an RTS, such that their execution will certainly not result in a quality degradation.

Observe that for items (2) and (4) in the above list of calibration operations, especially the predictive modeling required for the final platform is becoming a major source of uncertainty. That is due to the ongoing deep-submicron technology scaling which introduces strong process variability and reliability (degradation due to aging) effects. These effects can only be calibrated for at run-time, and the procedures described above allows for that.

For our H.264 example, a non-critical periodic calibration creates new scenarios when so far unseen frame macroblock breakups appear at run-time, and for each scenario sets the frequency-voltage pair based on the measured computation cycles required to decode that frame. When a given maximum number of scenarios has been reached, each time a new scenario needs to be introduced, one of the earlier introduced scenarios is removed again (using some heuristic to select that scenario). Moreover, another non-critical periodic calibration checks for the case when a large difference in computation cycles appears between the amount reserved for a scenario and the measured amount for all RTSs characterized to be in that scenario, so the frequency-voltage pair is modified to reduce the system energy consumption. However, when this adapted scenario introduces missed deadlines, the pair is reset to its initial value by a small adaption that is run for each RTS.

4. CASE STUDIES

In this section, we present an overview of the case studies we performed and which fit within the umbrella of the scenario methodology. Table II summarizes their contributions in each of the methodology steps. As all of these case studies were already published, in this paper, we emphasize only how the scenarios were defined, identified and used to reduce the system cost in each specific case. For each case, we show also how effective our approach was, by quantifying the final system improvements.

Step \ Case study	4.1	4.2	4.3	4.4
Identification	X	X	X	X
Prediction	X	x	x	X
Exploitation	x	X	X	X
Switching	x	x	x	x
Calibration	X	∼	∼	x

Legend

X	large contribution
x	small contribution
∼	not implemented

Table II. Methodology steps: implementation in case studies.

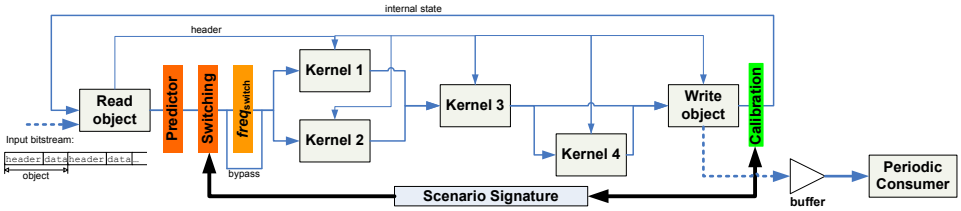

Fig. 13. Final implementation for the streaming application.

4.1 Energy Reduction via a Proactive DVFS-aware Scheduler

In [Gheorghita et al. 2008b], we have presented a general toolflow that can be applied to streaming applications to reduce their energy consumption by taking advantage of the DVFS mechanism available in modern processors. In this work, we aim at applications that are often implemented as a main loop, called the loop of interest, that is executed over and over again, reading, processing and writing out individual stream objects (e.g., the H.264 video decoder presented in figure 4). Each iteration of the loop of interest has a time constraint due to the throughput required by the system (usually specified by a standard). The actions executed during a loop iteration form an RTS. If in the beginning of each iteration of the loop of interest the number of required computation cycles is known, this information can be exploited to switch the processor frequency to the level that delivers the right performance to process the streaming object just in time.

To identify the RTS parameters, our toolflow uses the tool presented in section 3.3.1. The scenarios are derived by clustering the identified RTSs using as a cost function the required cycle budget. Besides taking into account all the important ingredients, like for example how often scenarios occur at run-time and the introduced cycle-budget over-estimation within a scenario, we consider for our clustering also the fact that the processor supply voltage/frequency cannot be changed continuously within the operation range, but only to some discrete points mentioned in its datasheet. Using the derived scenarios, our toolflow generates a predictor based on a decision diagram, as presented in section 3.4.

The structure of the final implementation of the application generated by our tool is shown in figure 13. The switching mechanism introduced into the application checks if it is necessary and energy efficient to switch from the current processor frequency to the one corresponding to the predicted scenario. When the required frequency is higher than the current one, a switch always occurs; when the frequency is lower, the application will switch only when the switching overhead is smaller than the energy saved by running the next RTS at this lower frequency. The calibration mechanism used in the application follows the structure given in figure 12. It changes the required cycle budget for a scenario when too many deadlines are missed for that scenario. Moreover, it also tunes for energy at run-time, as it implements all calibration situations presented in section 3.7. The calibration mechanism adds new scenarios for the cases that lead to the backup scenario being used, it defines different backup scenarios for different parts of the decision diagram, and it reduces over-estimation for scenarios for which the reserved budget is too high very often.

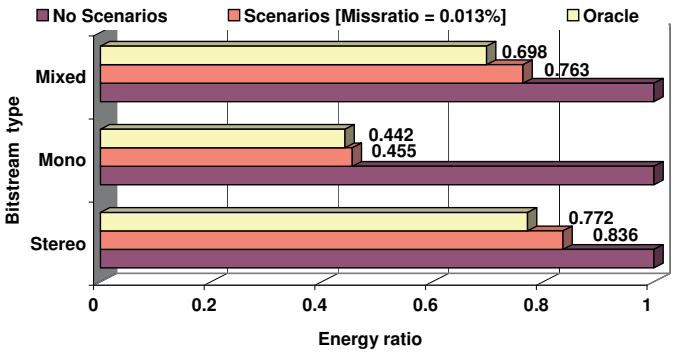

Fig. 14. Normalized energy reduction for an MP3 Decoder.

Figure 14 shows the energy reduction that we obtained when applying our trajectory to an MP3 decoder running on an Intel XScale PXA255 processor [Intel Corporation 2003]. The identification tool described in section 3.3.1 reduced the number of potential parameters ξ from all program variables to 41. This is done by incrementally profiling a training bitstream of increasing length while removing variables that can be assumed to have little effect on the program behavior. The result is a collection of 2111 potential scenarios. Using clustering algorithms as outlined in [Gheorghita et al. 2008b], we experimentally analyzed 34 different ways of splitting the application into scenarios to estimate the potential benefit of each of these options. The best obtained result contains 17 scenarios. The scenario identification process is time consuming, in particular the profiling based RTS parameter selection and the experimental evaluation of the quality of potential scenario sets, but it has been fully automated. The energy reduction obtained is 16% for stereo and 24% for a mixed set of stereo and mono streams respectively, for a miss ratio of up to one frame per 3 minutes (0.013%)[3]. The obtained energy improvement represents more than 72% of the maximum theoretically possible improvement (as visualized by the oracle bars in figure 14).

When omitting calibration, the energy reduction for the mixed set of stereo and mono streams evaluated in the case study drops from 24% to 16%. This shows the added value of calibration. A more detailed analysis of the experimental results shows that calibration to some extent compensates for the fact that the training bitstream used for RTS parameter discovery was insufficiently representative. The training bitstream was composed by taking some fragments from some random songs taken from internet. The results of this case study show that calibration increases the robustness of the method, alleviating the problem of creating representative input for parameter discovery.

We obtained similar results as those obtained for the MP3 decoder for two other applications, the motion compensation task of an MPEG-2 decoder and a G.72x voice decompression algorithm.

[3]These results differ from those presented in [Gheorghita et al. 2008b], because we used a different benchmark with less mono songs for the current experiments, and because the results of [Gheorghita et al. 2008b] were obtained without the run-time calibration for energy optimization.

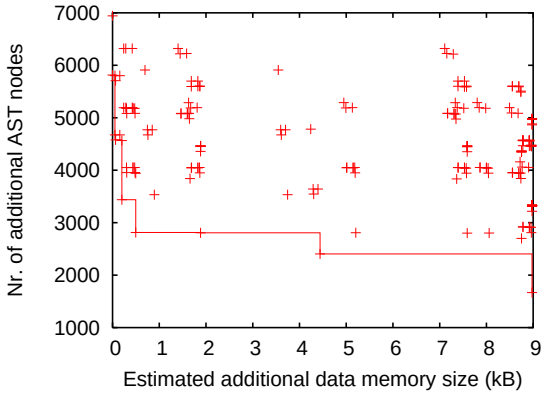

Fig. 15. Tradeoff between additional data memory size and instruction memory size (estimated by the nr. of additional AST nodes) for an MP3 Decoder.

4.2 Memory Access Optimization

In [Palkovic et al. 2005; Palkovic et al. 2006], we have presented an instantiation of our scenario methodology in the form of a general toolflow that can be applied to streaming applications to reduce their energy consumption by applying the scenario concepts on top of the Data Transfer and Storage Exploration (DTSE) methodology, particularly to enable more loop transformations in the DTSE. This work targets the same type of applications as the flow presented in section 4.1.

Using the identified RTS parameters, the RTSs corresponding to all possible different paths within the program's main loop are determined. The scenarios were derived by clustering the RTSs based on the possibility to apply similar loop transformations to the scenario as those that can be applied to the individual RTSs within this scenario and based on the code overhead of the scenario. We also took into account occurrence frequencies of individual RTSs resulting from the profiling. Thus, design-time static analysis of the available loop transformations and the corresponding code overhead was supplemented with profiling information. The six most frequently occurring RTSs (out of 234 RTSs in total) were grouped in potential scenario sets for further evaluation.

During the exploitation, the DTSE methodology is applied on each scenario from any selected potential scenario set, resulting in different source codes per scenario. The output of our scenario based loop transformation technique is a Pareto boundary (figure 15) which trades off additional data memory size and instruction memory size (estimated by the number of the Abstract Syntax Tree (AST) nodes in the resulting application code). Each point in the boundary corresponds to one scenario set with certain optimization potential and code size overhead. In the figure, all depicted non-Pareto points were obtained by a full exploration of all 203 options to cluster the six considered RTSs in scenarios. A DTSE optimized version of the original code was kept as a backup scenario for RTSs not part of the selected scenarios.

Because for our example the RTS parameter values can be read before the RTS itself occurs and because the scenario set is small, we do not need to use any

Source code version	Main data memory accesses	Improvement
Original	714.2×10^6	-
After DTSE	126.9×10^6	82.2%
After Scenarios and DTSE	68.8×10^6	90.3%

Table III. Comparison of different implementations of an MP3 audio decoder.

advanced predictor techniques. We use a simple scenario detector implemented as a lookup table, which based on the obtained RTS parameter values selects the appropriate (loop transformed) scenario. The scenario switching corresponds to loading the given scenario to the on-chip program memory. To introduce calibration mechanisms in the toolflow is not so simple, because the loop transformations on scenarios happen during design-time. We could of course perform the optimizations also during run-time, but the danger is that in such a way we do not meet many real-time deadlines.

For our experiments, we used a memory subsystem consisting of 2kB L1 on-chip data scratch-pad memory and off-chip main memory. Table III compares the number of main memory data accesses for three implementations of the MP3 code: original code, code after applying DTSE methodology and code after applying scenario+DTSE methodology (with three scenarios and a backup scenario). The DTSE methodology can reduce the number of main memory data accesses by 82.2%. However, with the scenario methodology on top of the DTSE, we can reduce the number of main memory data accesses by an additional 45.8% (8.1 percentage points with respect to the original application).

4.3 Dynamic Memory Management Refinement

In this subsection, we present the case study of optimizing dynamic memory allocation (i.e., MALLOC()/FREE()) for the IPv4 layer in an IEEE 802.11b wireless network application (for more details see [Mamagkakis et al. 2007]). In the first step of the scenario methodology, we identify all the RTSs, which consist of the packet sizes that are communicated and buffered in the IPv4 layer. We have identified 1460 different packet sizes (i.e., 1460 different RTSs) that can be buffered, but not all of them get buffered with the same frequency. As can be seen in figure 16, we have used 18 different network traces, of 1,000,000 packets each, to profile the wireless application. We concluded that the packet sizes that get buffered most often are the ACK packet size (= 40 Bytes) and the MTU packet size (= 1500 Bytes). Therefore, we decided to cluster the RTSs in three main system scenarios, according to their frequency distribution. This means that the predictor created in the second step of the scenario method will choose between 40 Byte-packets, 1500 Byte-packets and variable-Byte packets.

Exploiting the system scenarios in the third step, we designed a customized dynamic memory allocator which uses three system scenarios, each with one or more different pools of memory blocks, see figure 17. According to the buffer request of the wireless application, in the fourth step of the method, we switch to a different dynamic memory allocation strategy and thus resource usage of the embedded system is minimized. The selection algorithm is implemented in C++ with a single linked list and using simple if/elsif statements to select the appropriate pool of

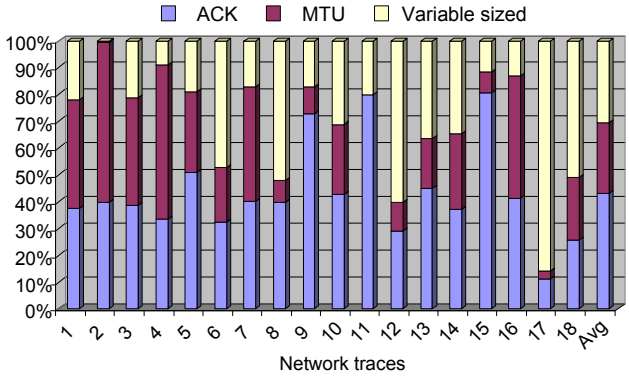

Fig. 16. Frequency distribution for the identification and prediction of three main system scenarios (i.e., for ACK packet size, for MTU packet size and for variable packet size)

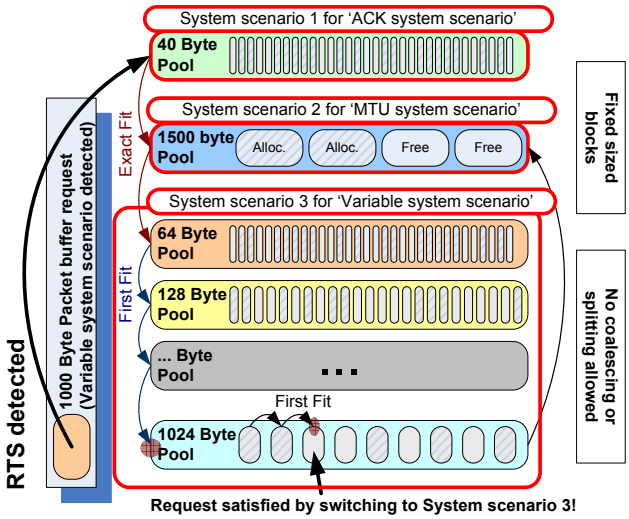

Fig. 17. Customized dynamic memory allocator design, which exploits and switches between three system scenarios (in the example the 'Variable system scenario' is detected)

memory blocks. As can be seen in figure 17, a request to buffer a 1000 Byte packet is detected and thus the request is satisfied by switching to the third system scenario. With the use of our dynamic memory allocator, which is customized to detect and exploit the most probable system scenarios, we manage to achieve on average a reduction of 90% for energy consumption, of 62% for memory footprint and an 81% improvement of performance compared to the dynamic memory allocator used in the Linux operating system. The scenario switching (step 4) is inherently embedded in the design of the run-time dynamic memory manager [Mamagkakis et al. 2007].

4.4 High Performance Processor Adaptation

In another study [Vandeputte et al. 2007], we have used the scenario approach to adapt the hardware configuration of high performance processors for energy efficiency. When a program is being executed on a high performance processor, it typically does not need all available processor resources at the same time. The amount of resources that a program needs actually depends on the code that is being executed and typically varies over time [Sherwood et al. 2002]. By switching off the processor resources that are not needed during the execution of a program, the energy consumption of the processor can be reduced substantially while limiting the performance degradation.

At design-time, we extract a scenario set containing the different phases of a given program by profiling the program. A program phase is the set of all parts of the execution of a program that exhibit similar behavior. In our case, each scenario corresponds with one program phase; the scenario set thus simply consists of all program phases.

To extract the set of phases for a given program, we divide the execution of a program into non-overlapping instruction intervals of a given fixed length. Each instruction interval thus forms one RTS. For each instruction interval, we collect a Basic Block Vector (BBV) [Sherwood et al. 2002], which is a weighted frequency vector that captures which basic blocks of the program code have been executed during the corresponding instruction interval. These BBVs are the RTS parameters of our framework. We then group similar instruction intervals (i.e., RTSs) into one phase (scenario) by clustering the BBVs through K-means clustering. During the clustering process, we also take into account the predictability of the given scenario set. Once we have extracted the scenario set, we then identify the energy-optimal processor configuration for each individual scenario through an efficient offline configuration space exploration algorithm.

At run-time, the per-phase optimal hardware configurations are first communicated from software to hardware. The adaptive processor collects BBV-information for each instruction interval. At the end of each instruction interval, the hardware predicts if the running scenario will continue for at least one more interval or if the program will switch to another scenario during the next interval. For this prediction, we use a Markov-based predictor [Vandeputte et al. 2005]. If a scenario switch is predicted, the configuration of the processor is adapted to the energy-optimal hardware configuration of the predicted scenario. If the program is executing a program phase that has not been captured at design-time, a back-up scenario is used, meaning that the default processor configuration is being used.

We evaluated our approach on an aggressive 8-issue out-of-order superscalar processor microarchitecture with an adaptive branch predictor, processor width, reorder buffer, fetch buffer and caches. These microarchitectural parameters are the knobs in our system. The entire configuration space consists of 10^{15} processor configurations or knob positions. We evaluated this technique on the SPEC CPU2000 benchmarks. The results are shown in figure 18. On average, our hardware adaptation approach achieves a 36% reduction in energy consumption with a performance (Instructions Per Cycle or IPC) degradation of only 2.9%.

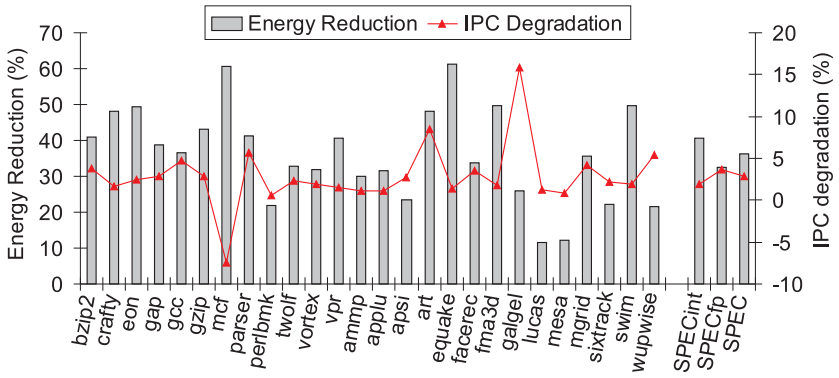

Fig. 18. Energy reduction and performance penalty.

4.5 Other Approaches

Besides the already presented case studies, in our research groups other approaches have been developed at various levels of design abstraction, which are relevant and demonstrate the proposed scenario-based methodology from many different angles.

The scenario concept was first used in the context of multi-task applications [Yang 2004; Yang et al. 2002] to capture the data-dependent dynamic behavior inside a thread, to better schedule a multi-thread application on a heterogenous multi-processor architecture, allowing the change of voltage level for each individual processor. The work also includes a system scenario based DVFS hybrid design-time/run-time scheduler technique. However, the scenario identification and run-time predictor are done manually.

In [Sanz et al. 2006], the energy and performance efficiency of memories is increased using scenario-based methodologies that tackle process variability problems. The paper shows that apart from the application RTSs, another significant source of unpredictability is the platform itself when using technology scaling beyond the 90nm technology node.

In [Bougard et al. 2006], it is illustrated that the design of wireless systems benefits from the scenario concept by characterizing wireless channels and user conditions as RTS parameters. The predicted scenarios are used to exploit cross-layer (in the OSI model) tradeoffs and switch between knob positions, adjusting the user throughput and system power accordingly at run-time. Also in the wireless domain, the work presented in [Pollin et al. 2007] manages to reduce energy consumption considerably by exploiting run-time controllable knobs of actual RF components and the 802.11 Medium Access Controller.

In [Peon-Quiros et al. 2007], a methodology is introduced that identifies scenarios of wireless mobile streams in multi-threaded applications. The RTSs consider the change in size among packets in a wireless network stream. The identified scenarios are, then, used to control at run-time a knob that switches DMA block transfers of heap data on and off and saves energy and execution cycles accordingly. In [Yang et al. 2001], the scenario concept is used for energy-efficient scheduling of tasks

for heterogeneous MPSoC embedded platforms. The scenarios are extracted from the task graph of the software applications and used to reduce energy through the proposed DVFS methodology.

In [Poplavko et al. 2007], a method for estimating the execution time of stream-oriented applications mapped on a multi-processor on-chip is detailed. For this type of systems, the pipelined decoding of sequential streaming objects has a high impact on achieving the required throughput. The application is modeled as a homogenous synchronous data flow graph (HSDF). Within the application's loop of interest the scenarios are manually defined based on the different execution workloads of tasks. An accurate execution time estimation method is proposed that supports parallel and pipelined decoding of streaming objects, taking into account the transient and periodic behavior of scenario executions and the effect of scenario transitions. The method can be applied in quality-of-service or resource management contexts. In [Hamers and Eeckhout 2007], scenarios are used to allow prediction of decode time, decode energy, and quality-of-service of media streams in a client-server quality-of-service management context. The proposed resource prediction method runs on the content-provider side. The results can be used at the client side to determine the desired resource usage-quality tradeoff. A resource-constrained, battery-powered handheld can for example choose to lower the resolution of a video stream to guarantee a battery life time that is sufficient to view the entire stream.

A more general model than the HSDF model used in [Poplavko et al. 2007] that is still analyzable is the scenario-aware data flow model (SADF) [Theelen et al. 2006]. It is a design-time analyzable stochastic generalization of the synchronous data flow (SDF) model, which can capture several dynamic aspects of modern streaming applications by incorporating system scenarios. The scenarios and the run-time predictor are explicitly described in the model, no further need for identification of scenarios for applications written using this model being necessary. Moreover, analysis of long-run average and worst case performance are decidable. SADF combines both analyzability and explicit representation of scenarios. The only current drawback is that not all possible forms of dynamism (e.g., interactions with external events) can be represented with it.

The DVFS application of the system scenario concepts used throughout the paper and detailed in section 4.1 assumes that timing constraints are soft. As already mentioned, soft requirements allow to trade off quality and cost. In [Gheorghita et al. 2005], we illustrate a DVFS application that provides hard guarantees. Scenario identification is based entirely on a conservative static analysis. Also the method of [Poplavko et al. 2007] described above is suitable to provide hard guarantees. In general, it may not always be easy to make all steps of the scenario methodology conservative, but whenever this is possible our methodology applies, omitting calibration which is not meaningful in a context requiring hard guarantees.

At a higher design abstraction level, in [Daylight et al. 2002], scenarios for memory management of the heap data are defined by the user using the steady-state concept. The identified scenarios are used to switch between dynamic data types according to the predicted scenarios in multimedia software applications. In [Temmerman et al. 2007], the design abstraction level is raised even higher and scenario-based optimizations are made at the modeling level. In this paper, scenarios are

identified based on behavioral information and different UML transformations are proposed for the abstract data types of multimedia applications in order to achieve tradeoffs between energy, memory accesses and memory footprint.

Finally, a variant of the scenario concept with a much bigger number of RTSs to accommodate even more dynamic situations has been used in [Tack et al. 2006] for 3D graphics applications. It considers so-called sub-scenarios that exhibit specific correlated behavior and that are characterized individually at design-time, but that are merged only at run-time.

5. RELATED WORK

This section consists of two parts. The first part compares our system scenario based methodology with related approaches, while the second part presents exploitation examples of scenarios found in the literature (not related to our own efforts, which were already surveyed in the previous section).

5.1 Related Design Approaches

In the past, embedded system design was significantly improved using the inspector-executor technique, which was developed at University of Maryland in the early 1990ties [Saltz et al. 1991]. The basic idea behind it is to compile the application loops in two phases, an inspector and an executor. The inspector examines the data access pattern in the loop body and creates a schedule for fetching the values stored in remote memories. The executor retrieves remote values according to the schedule and executes the loop. The authors have studied run-time methods to automatically parallelize and schedule iterations of a loop in certain cases when compile-time information is inadequate. At compile-time, these methods set up the framework for performing a loop dependency analysis. At run-time, wavefronts of concurrently executable loop iterations are identified and the loop iterations are reordered for increased parallelism. A similar approach has been taken also in [Arenaz et al. 2004] where a loop with irregular assignment computations contains loop-carried output data dependencies that can only be detected at run-time. A load-balanced method based on the inspector-executor model is proposed to parallelize this loop pattern. The basic idea lies in splitting the iteration space of the sequential loop into sets of conflict-free iterations that can be executed concurrently on different processors. In [Yokota et al. 2002], the authors propose a modified inspector-executor method for implementing accesses to a distributed array. In the method, the compiler runs an inspector during compile time to obtain the information of data dependencies among node processors, and it uses that information to optimize communication code included in the executor. In [van der Mark et al. 2004], a novel strategy is discussed, which dynamically drives the communication between the processors by examining the content of the data at run-time in order to reduce communication costs for numerical weather prediction modes. Compared to the inspector-executor which is based on low-level data access patterns, this strategy includes high-level application dependent information.

System workload characterization is another related field of research. It is particulary relevant for the scenario identification step of our methodology. It gained interest already more than 30 years ago [Ferrari 1972]. First, it has been used for selecting the appropriate workload for doing meaningful measurements on the per-

formance of computer systems. Later, workload characterization has been extended to wired [Lee 1991] and wireless [Kotz and Essien 2005] networks. Moreover, it also was considered as a base for traffic shaping which is used for adapting the workload to the expected workload in the network/application [Raman and Chakraborty 2006]. A specific area in workload characterization is the identification of program phases [Sherwood et al. 2002]. Programs usually consist of a number of repeating execution patterns, which are identified. In the program phase detection, code-based phase detection techniques [Huang et al. 2003] and interval-based phase detection techniques [Sherwood et al. 2002] are used. In code-based phase detection, program phases are associated with functions and loops. The interval-based phase detection techniques divide the execution of a program into fixed-length instruction intervals and group intervals with similar characteristics. A detailed survey about workload characterization can be found in [Calzarossa and Serazzi 1993]. It identifies five common steps followed by all workload characterization approaches, including our scenario identification techniques: (i) choice of the set of parameters able to describe the behavior of the workload, (ii) choice of the suitable instrumentation, (iii) experimental measurement collection, (iv) analysis of the workload data, and (v) construction of workload models.

Workload characterization and the inspector-executor technique perform most of the analysis at run-time. This approach is beneficial, when design-time analysis is not available. The system scenario methodology for designing embedded systems is more general in the sense that it can handle systems with unpredictable and extremely varying workloads where the previous techniques cannot be used. The system is made more predictable via design-time analysis. The actual behavior of the system, obtained by combining static analysis and profiling approaches, is split into distinct classes (scenarios) of typical workload behavior. System scenarios allow optimization of the system mapping for each scenario, optimizations from which the system profits when the scenario appears at run-time. This combination of design-time analysis and classification of behaviors with run-time exploitation and potentially calibration is the main novelty of the scenario based approach.

Due to the presence of the run-time calibration step in our methodology, the scenario approach is related to adaptive controllers [Dumont and Huzmezan 2002]. However, the scenario approach distinguishes itself via the design-time preparation and classification of system behaviors, which guides the calibration into the most promising directions (by pruning directions that are known to be of no interest). Furthermore, for cost reasons, at run-time, our calibration technique is only active at certain designated moments in time (calibration time) whereas a typical adaptive controller executes continuously.

The system scenario concept was identified explicitly for the first time in [Yang et al. 2002], where it was used to improve the mapping of dynamic applications onto a multi-processor platform. Concepts closely related to the scenario idea already appear in [Marchal et al. 2000]. Our scenario methodology ideas were for the first time briefly introduced in [Gheorghita et al. 2006]. That paper does not detail any of the methodology steps, and even entirely omits the switching and calibration steps. It also contains only one case study and a much less extensive literature survey.

5.2 Scenario Exploitation Examples

Scenario-like concepts were applied in an ad-hoc manner several times, with an emphasis on exploiting scenarios, and not on identifying and predicting them. In [Chung et al. 2002], the authors use in a systematic way the information about periodicity of multimedia applications to present a new concept of DVFS. Each period in the application shows a large variation in terms of execution time. The proposed idea is to supply the information of the execution time variations in addition to the content itself. This makes it possible to perform DVFS independent of worst case execution time estimation providing energy consumption reduction of client systems compared to previous DVFS techniques. However, the authors do not specify how the periods should be identified. In [Sasanka et al. 2002], for each manually identified scenario, the authors 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 run-time. In [Choi et al. 2002], a reactive predictor is used to select the lowest supply voltage for which the timing constraints of an MPEG decoder are met. An extension [Sachs et al. 2003] considers two simultaneous resources for scenario characterization. It looks for the most energy efficient configuration for encoding video on a mobile platform, exploring the tradeoff between computation and compression efficiency.

In the context of multi-task applications, in [Murali et al. 2006a], scenarios are characterized by different communication requirements (such as different bandwidth, latency constraints) and traffic patterns. The paper presents a method to map an application to a network on chip (NoC) architecture, satisfying the design constraints of each individual scenario. This approach concentrates on the communication aspect of the application mapping. It allows dynamic network reconfiguration across different scenarios. As the over-estimation of the worst case communication is very large, this method performs poorly on systems where the traffic characteristics of scenarios are very different or when the number of scenarios is large. In [Murali et al. 2006b], the method was extended to work for these cases too.

Besides the already mentioned applications of scenario concepts in the context of the HSDF and SADF models of computation, in [Lee et al. 2002], a combination of a hierarchical finite state machine (FSM) with a synchronous data flow model (SDF) is used to capture scenarios within a multi-task streaming application. The FSM represents the scenarios' run-time detector. The scenarios are identified by the designer and they are already described in the model. The authors showed that by writing the application in this model, the scenario knowledge can be used to save energy when mapping the application on one processor. The SADF model of computation of [Theelen et al. 2006] generalizes this FSM-SDF model of computation.

Another example of improving a multi-task application analysis approach using application scenarios is [Wandeler and Thiele 2005]. This paper extends an existing method for performance analysis of hard-real-time systems based on Real-Time Calculus, taking into account correlations that appear between different compo-

nents of the system. The knowledge about these correlations is used to derive the system scenarios. The authors present only how these scenarios could be modeled in their high level modeling/analytical approach, but no way to identify scenarios and no prediction mechanism was considered.

As a final observation, quality of service (QoS) mechanisms may make use of scenarios and related concepts. However, a detailed discussion of QoS techniques is beyond the scope of this paper. More information can be found in papers related to QoS, like [Goossens et al. 2003; Vogel et al. 1995].

In summary, compared to the previous work on workload characterization and on the inspector-executor approach, which target a purely run-time approach, this paper targets a combined design-time and run-time approach. The previous work in system scenarios is either quite ad-hoc, e.g., [Chung et al. 2002; Sasanka et al. 2002; Murali et al. 2006b], or targets specific contexts, e.g., [Yang et al. 2002; Gheorghita et al. 2008b; Palkovic et al. 2006; Hamers et al. 2007]. In this paper, we propose a general systematic scenario methodology, outlining generally applicable solutions for the various steps whenever possible. The method can be instantiated for a wide variety of specific goals, as illustrated in section 4.

6. CONCLUSIONS

In this paper, we introduced the concept of *system scenarios*, that cluster the Run-Time Situations (RTSs) in which a system may run based on similarities from a cost perspective (e.g., resource utilization), in such a way that a system can be optimized at design-time and configured at run-time to exploit this cost similarity. Different from the well known use-case scenarios, which are manually written diagrams that represent the user perspective on future system usage, system scenarios can be automatically derived, are transparant to the user, and focus on system cost optimization.

A combined design-time/run-time methodology for using system scenarios to improve the final system cost is detailed. At design-time, the scenarios in the system are identified and each of them is exploited by applying different and more aggressive optimizations. The scenarios are combined together in the final system, with a prediction, a switching and, potentially, a calibration mechanism. These mechanisms have important roles at run-time. Prediction finds out in advance of the RTS execution in which scenario the system will run, and using the switching mechanism the appropriate scenario is set enabling the optimizations applied for that specific scenario at design-time. The calibration mechanism allows the system to learn on-the-fly how to further reduce its final cost, by adapting to the current environment (e.g., system temperature, input data). The operations done by the calibration include extending (or reducing) the scenario set, modifying the scenario definitions, and changing both the prediction and switching mechanisms. Calibration can also be applied to preserve system quality, e.g., when scenarios cause unexpected or incidental high overheads, or to trade off quality and cost. Our system scenario based methodology can be integrated within existing embedded system design flows, to increase their performance in reducing the cost of the resulting systems, while maintaining their quality.

Besides the general method, the paper outlines general and coherent solution

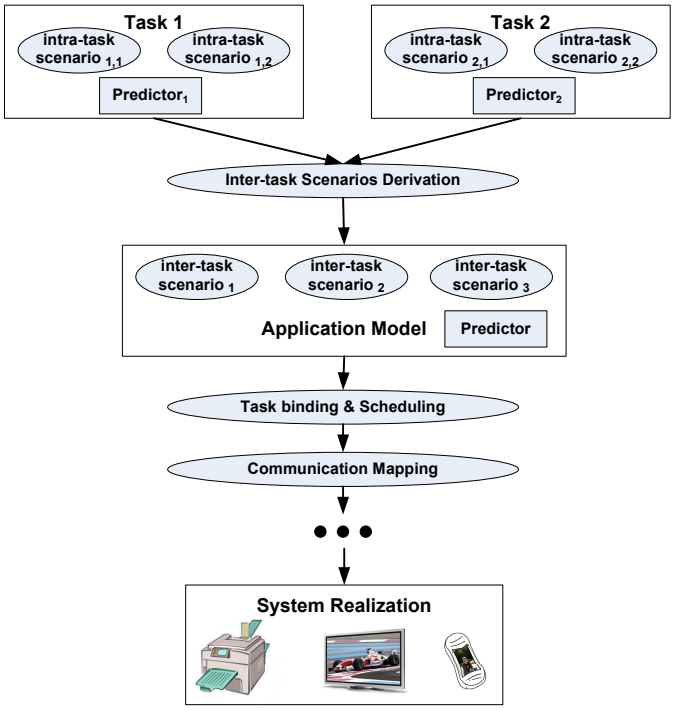

Fig. 19. Required design flow for multi-task multi-processor systems.

strategies for some of its core concepts, namely scenario identification, prediction, and calibration. Furthermore, four case studies are presented to show how the methodology was applied to several real-life design problems. The diversity of these case studies, and the other given exploitation examples, emphasizes the fact that, although the various design problems look different, they can be covered by the same methodology. The obtained reductions in the final system costs prove that applying our methodology in the design process leads to better products.

Although some of the surveyed applications of the system scenario concepts target multi-task or multi-processor systems, the use of scenarios within a multi-processor embedded system design trajectory has not been extensively explored yet. Given the importance of multi-processor systems, we consider this as an interesting direction for future research. A design flow like the one sketched in figure 19 can be envisioned. The flow starts with extracting intra-task scenarios for each application task, and based on them derives the inter-task application scenarios. The intra- and inter-task scenarios are conceptually the same from a methodology perspective, so that RTS parameter discovery techniques and scenario prediction techniques can be reused. However, intra- and inter-task scenarios have a different impact on the intra- and inter-task parts of the remaining design trajectory, their exploitation being in general different. Certain parts of the scenario methodology are affected. Characterizing the cost (e.g., the cycle budget) of an RTS, which is part of scenario identification, has to be adapted to accommodate the specific problems that appear

in multi-task applications, like, intra- and/or inter-processor scheduling, inter-task communication costs and delays, pipelined execution. These problems make the resource estimation for multi-task applications, especially in a multi-processor context, a challenging research topic. The accuracy of resource estimation does not only affect scenario identification, but also calibration. It may furthermore be necessary to distribute calibration or to collect calibration information at a central location in the system. After deriving inter-task system scenarios, they are used in decision making along the design trajectory, like in task binding and scheduling. Depending on the type of exploitation, various switching mechanisms may be needed, varying from configuring simple settings to migrating tasks from one processor to another one. Advanced switching mechanisms such as run-time task migration are another interesting research direction. Finally, if multiple scenario-aware applications can coexist in the same multi-application system, scenario-aware resource and quality of service management across applications needs to be investigated.

Acknowledgements. This work was supported in part by the Dutch Science Foundation, NWO, project FAME, number 612.064.101, by the Fund for Scientific Research in Flanders, FWO, projects G.0160.02 and G.0255.08, by Ghent University via a BOF grant, and by the European Commission, through Marie Curie project DACMA MEST-CT-2004-504767 and the Artist and HiPEAC Networks of Excellence. Lieven Eeckhout is a Postdoctoral Fellow with FWO. We thank the anonymous reviewers for their useful feedback.

REFERENCES

Arenaz, M., Touriño, J., and Doallo, R. 2004. An inspector-executor algorithm for irregular assignment parallelization. In *2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2004)*. Hong Kong, China, 4–15.

Bougard, B., Dejonghe, A., and Dehaene, W. 2006. Cross-layer power management in wireless networks and consequences on system-level architecture. *Signal Processing 86,* 8, 1792–1803.

Burd, T., Pering, T., Stratakos, A., and Brodersen, R. 2000. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits 35,* 11 (November), 1571–1580.

Calzarossa, M. and Serazzi, G. 1993. Workload characterization: a survey. *Proceedings of the IEEE 81,* 8, 1136–1150.

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

Catthoor, F., Ed. 2000. *Unified Low-Power Design Flow for Data-Dominated Multi-Media and Telecom Applications*. Kluwer Academic Publishers, Boston, MA.

Choi, K., Dantu, K., Cheng, W.-C., and Pedram, M. 2002. 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, 732–737.

Chung, E.-Y., De Micheli, G., and Benini, L. 2002. Contents provider-assisted dynamic voltage scaling for low energy multimedia applications. In *Proceedings of the 2002 international symposium on Low power electronics and design (ISLPED'02)*. ACM Press, New York, NY, USA, 42–47.

Daylight, E. G., Wuytack, S., Ykman-Couvreur, C., and Catthoor, F. 2002. Analyzing energy friendly steady state phases of dynamic application execution in terms of sparse data structures. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED'02)*. ACM Press, New York, NY, 76–79.

De Sutter, B., De Bus, B., and De Bosschere, K. 2006. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems 27,* 5, 882–945.

Debray, S., Evans, W., Muth, R., and De Sutter, B. 2002. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems 22,* 2, 378–415.

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

Dumont, G. A. and Huzmezan, M. 2002. Concepts, methods and techniques in adaptive control. In *Proc. of the American Control Conference (ACC)*. Vol. 2. IEEE, 1137–1150.

Ferrari, D. 1972. Workload charaterization and selection in computer performance measurement. *Computer 5,* 4, 18–24.

Fowler, M. 2003. Use cases. In *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison Wesley Publishing Company, Reading, MA, Chapter 9, 99–106.

Geilen, M. C. W., Basten, T., Theelen, B. D., and Otten, R. H. J. M. 2007. An algebra of pareto points. *Fundamenta Informaticae 78,* 1, 35–74.

Gheorghita, S. V., Basten, T., and Corporaal, H. 2005. 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, 177–184.

Gheorghita, S. V., Basten, T., and Corporaal, H. 2006. Application scenarios in streaming-oriented embedded system design. In *Proc. of the International Symposium on System-on-Chip (SoC 2006)*. IEEE Press, Piscataway, NJ, 175–178. Revised version to appear as [Gheorghita et al. 2008a].

Gheorghita, S. V., Basten, T., and Corporaal, H. 2008a. Application scenarios in streaming-oriented embedded system design. *IEEE Design & Test of Computers*. Invited. Best paper of SoC 2006. To appear.

Gheorghita, S. V., Basten, T., and Corporaal, H. 2008b. Scenario selection and prediction for DVS-aware scheduling. *Journal of Signal Processing Systems 50,* 2, 137–161.

Gheorghita, S. V., Stuijk, S., Basten, T., and Corporaal, H. 2005. Automatic scenario detection for improved WCET estimation. In *Proc. of the 42nd Design Automation Conference DAC*. ACM Press, New York, NY, 101–104.

Goossens, K., Dielissen, J., van Meerbergen, J., Poplavko, P., Radulescu, A., Rijpkema, E., Waterlander, E., and Wielage, P. 2003. Guaranteeing the quality of services in networks on chip. In *Networks on chip*. Kluwer Academic Publishers, Hingham, MA, USA, Chapter 4, 61–82.

Hamers, J. and Eeckhout, L. 2007. Resource prediction for media stream decoding. In *Proc. of Design, Automation and Test in Europe (DATE)*. IEEE, 594–599.

Hamers, J. and Eeckhout, L. 2008. Exploiting media stream similarity for energy-efficient decoding and resource prediction. *ACM Transactions on Embedded Computing Systems*. To appear.

Hamers, J., Eeckhout, L., and De Bosschere, K. 2007. Exploiting video stream similarity for energy-efficient decoding. In *Proc. of the 13th International Multimedia Modeling Conference, (MMM)*. LNCS, vol. 4352. Springer, Berlin, Germany, 11–22. Extended version to appear as [Hamers and Eeckhout 2008].

Hansson, A., Coenen, M., and Goossens, K. 2007. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. of Design, Automation, and Test in Europe (DATE)*. IEEE Press, Piscataway, NJ, 954–959.

Huang, M., Renau, J., and Torrellas, J. 2003. Positional adaptation of processors: Application to energy reduction. In *International Symposium on Computer Architecture (ISCA)*. IEEE CS Press.

Hughes, C. J., Srinivasan, J., and Adve, S. V. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proc. of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*. IEEE Computer Society, Washington, DC, 250–261.

IEEE. 2000. IEEE standard 1471: Recommended practice for architectural description of software-intensive systems.

Intel Corporation. 2003. Intel XScale microarchitecture for the PXA255 processor: Users manual. Order No. 278796.

Ionita, M. T. 2005. Scenario-based system architecting: a systematic approach to developing future-proof system architectures. Ph.D. thesis, Technische Universiteit Eindhoven, The Netherlands.

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

Kotz, D. and Essien, K. 2005. Analysis of a campus-wide wireless network. *Wireless Networks 11,* 1, 115–133.

Lee, R. 1991. An introduction to workload characterization. http://support.novell.com/techcenter/articles/ana19910503.html.

Lee, S., Yoo, S., and Choi, K. 2002. An intra-task dynamic voltage scaling method for SoC design with hierarchical FSM and synchronous dataflow model. In *Proc. of the International Symposium on Low Power Electronics and Design*. ACM Press, 84–87.

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

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

Murali, S., Coenen, M., Radulescu, A., Goossens, K., and De Micheli, G. 2006a. Mapping and configuration methods for multi-use-case networks on chips. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*. ACM Press, 146–151.

Murali, S., Coenen, M., Radulescu, A., Goossens, K., and De Micheli, G. 2006b. A methodology for mapping multiple use-cases onto networks on chips. In *Proc. of Design, Automation and Test in Europe (DATE)*. IEEE, 118–123.

Okabe, T., Jin, Y., and Sendhoff, B. 2003. A critical survey of performance indices for multi-objective optimisation. In *Proc. of the Congress on Evolutionary Computation*. Vol. 2. IEEE Press, Piscataway, NJ, 878–885.

Palkovic, M., Brockmeyer, E., Vanbroekhoven, P., Corporaal, H., and Catthoor, F. 2006. Systematic preprocessing of data dependent constructs for embedded systems. *Journal of Low Power Electronics 2,* 1 (April), 9–17.

Palkovic, M., Corporaal, H., and Catthoor, F. 2005. Global memory optimisation for embedded systems allowed by code duplication. In *Proc. 9th Intnl. Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 72–79.

Pareto, V. 1906. *Manuale di Economia Politica*. Piccola Biblioteca Scientifica, Milan. Translated into English by A.S. Schwier (1971), Manual of Political Economy, MacMillan, London.

Paul, J. M., Thomas, D. E., and Bobrek, A. 2006. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Transations on Very Large Scale Integration (VLSI) Systems 14,* 8, 868–880.

Peon-Quiros, M., Bartzas, A., Mamagkakis, S., Catthoor, F., Mendias, J., and Soudris, D. 2007. Direct memory access optimization in wireless terminals for reduced memory latency and energy consumption. In *Proc. of the 17th International Workshop in Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Springer, 373–383.

Pollin, S., Mangharam, R., Bougard, B., Van der Perre, L., Moerman, I., Rajkumar, R., and Catthoor, F. 2007. MEERA: Cross-layer methodology for energy efficient resource allocation in wireless networks. *IEEE transactions on wireless communications 6,* 2, 617–628.

Poplavko, P., Basten, T., and van Meerbergen, J. 2007. Execution-time prediction for dynamic streaming applications with task-level parallelism. In *Proc. of 10th EUROMICRO Conference on Digital System Design (DSD)*. IEEE Computer Society Press, 228–235.

Raman, B. and Chakraborty, S. 2006. Application-specific workload shaping in multimedia-enabled personal mobile devices. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 4–9.

Sachs, D. G., Adve, S. V., and Jones, D. L. 2003. Cross-layer adaptive video coding to reduce energy on general-purpose processors. In *Proc. of IEEE International Conference on Image Processing*. IEEE Press, 109–112.

Saltz, J. H., Mirchandaney, R., and Crowley, K. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Computers 40,* 5, 603–612.

Sanz, C., Prieto, M., Papanikolaou, A., Miranda, M., and Catthoor, F. 2006. System-level process variability compensation on memory organizations of dynamic applications: a case study. *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED)*, 376–382.

Sasanka, R., Hughes, C. J., and Adve, S. V. 2002. Joint local and global hardware adaptations for energy. *ACM SIGARCH Computer Architecture News 30,* 5, 144–155.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York, NY, 45–57.

Shin, D. and Kim, J. 2005. Optimizing intra-task voltage scheduling using data flow analysis. In *Proc. of the 10th Asia and South Pacific Design Automation Conference (ASPDAC)*. ACM Press, New York, NY, 703–708.

Tack, K., Lafruit, G., Catthoor, F., and Lauwereins, R. 2006. Platform independent optimisation of multi-resolution 3D content to enable universal media access. *The Visual Computer 22,* 8, 577–590.

Temmerman, M., Daylight, E. G., Catthoor, F., Demeyer, S., and Dhaene, T. 2007. Optimizing data structures at the modeling level in embedded multimedia. *Journal of Systems Architecture 53,* 8, 539–549.

Theelen, B. D., Geilen, M. C. W., Basten, T., Voeten, J. P. M., Gheorghita, S. V., and Stuijk, S. 2006. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proc. of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE Computer Society Press, 185–194.

van der Mark, P., Wolters, L., and Cats, G. 2004. Using semi-lagrangian formulations with automatic code generation for environmental modeling. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*. ACM, 229–234.

Vandeputte, F., Eeckhout, L., and De Bosschere, K. 2005. A detailed study on phase predictors. In *Proceedings of the 11th International Euro-Par Conference*, J. Cunha and P. Medeiros, Eds. LNCS, vol. 3648. Springer, 571–581.

Vandeputte, F., Eeckhout, L., and De Bosschere, K. 2007. Exploiting program phase behavior for energy reduction on multi-configuration processors. *Journal of Systems Architecture 53,* 8, 489–500.

Vogel, A., Kerherve, B., von Bochmann, G., and Gecsei, J. 1995. Distributed multimedia and QoS: a survey. *IEEE Multimedia 2,* 2 (April), 10–19.

Wandeler, E. and Thiele, L. 2005. Characterizing workload correlations in multi processor hard real-time systems. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society Press, 46–55.

Wegener, I. 2000. Integer-Valued DDs. In *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, Philadelphia, PA, Chapter 9.

Yang, P. 2004. Pareto-optimization based run-time task scheduling for embedded systems. Ph.D. thesis, Catholic University of Leuven, Belgium.

Yang, P. and Catthoor, F. 2003. Pareto-optimization-based run-time task scheduling for embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03)*. ACM Press, New York, NY, USA, 120–125.

Yang, P., Marchal, P., Wong, C., Himpe, S., Catthoor, F., David, P., Vounckx, J., and Lauwereins, R. 2002. Managing dynamic concurrent tasks in embedded real-time multimedia

systems. In *Proc. 15th ACM/IEEE Intnl. Symp. on System-Level Synthesis (ISSS)*. IEEE Computer Society, Los Alamitos, CA, USA, 112–119.

Yang, P., Wong, C., Marchal, P., Catthoor, F., Desmet, D., Verkest, D., and Lauwereins, R. 2001. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Des. Test 18,* 5, 46–58.

Ykman-Couvreur, C., Brockmeyer, E., Nollet, V., Marescaux, T., Catthoor, F., and Corporaal, H. 2005. Design-Time Application Exploration for MP-SoC Customized Run-Time Management. In *Proc. of the International Symposium on System-on-Chip (SoC)*. IEEE Press, Piscataway, NJ, 66–69.

Ykman-Couvreur, C., Nollet, V., Catthoor, F., and Corporaal, H. 2006. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *Proc. of the International Symposium on System-on-Chip (SoC)*. IEEE Press, Piscataway, NJ, 1–4.

Yokota, D., Chiba, S., and Itano, K. 2002. A new optimization technique for the inspector-executor method. In *International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*. Acta Press, 706–711.