

# Intra-task Scenario-aware Voltage Scheduling

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

## ABSTRACT

Modern embedded applications usually have real-time constraints and they have requirements for low energy consumption. At system level, intra-task dynamic voltage scaling (DVS) is one of the most effective techniques for energy reduction. It changes the processor's supply voltage and clock frequency to the lowest level that still allows the real-time constraints to be met. In this paper, we present how intra-task scenarios, which capture correlations between different parts of the application, can be applied on top of existing DVS techniques, making them more effective. Furthermore, we extend our method for automatic discovery of scenarios and adapt it to the DVS requirements. We show that, by augmenting an existing DVS method with scenarios, the average energy consumption of two real-life benchmarks is reduced with 14% to 52%.

**Categories and Subject Descriptors:** C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

**General Terms:** Algorithms, Performance, Design

**Keywords:** WCET, Dynamic Voltage Scheduling, Real-Time, Scenarios

## 1. INTRODUCTION

The increasing development of mobile embedded systems, like mobile phones, PDAs, digital cameras, has directed designers' interest in finding solutions for increasing the battery lifetime of these systems. The problem complexity increases when dealing with real-time systems, where, besides reducing the energy consumption and power dissipation of the entire system, there are also tight performance constraints to be met.

At system level, the most effective low-power techniques for real-time systems are dynamic voltage scaling (DVS) and dynamic power management (DPM) aware scheduling [9]. They take into account that the processor energy consumption depends quadratically on the supply voltage ( $E \propto V_{DD}^2$ ), while its execution speed (frequency) depends linearly on the supply voltage ( $f_{CLK} \propto V_{DD}$ ). By using DVS, different tasks or parts of a task run at different clock frequencies and supply voltage levels, while still providing the required performance. DPM [13] suspends system parts which

are not currently used, reducing their energy consumption. When both DVS and DPM are available for an architecture, it is known that it is always advantageous to exploit DVS first [9].

Depending on the granularity, there are two different approaches for DVS-aware scheduling: *inter-task voltage scheduling* [8, 21, 3] and *intra-task voltage scheduling* [11, 14, 19, 4, 16, 18]. The first one determines the voltage on a task basis, while the second one selects voltage levels within the task. In this paper, we present a method for improving the performance of existing intra-task scheduling algorithms. These algorithms exploit the slack time that appears at runtime because of the difference between the length of the worst case execution path and the current execution path. To do this, in some points of the original program, called *voltage scaling points*, a piece of code that may change the clock frequency based on the currently followed execution path of the program is inserted.

The energy consumption reduction depends on the amount of slack time and when it is observed during the runtime. The earlier it is detected, the more energy may be saved. Most of the current approaches are *reactive*: after a piece of code is executed, the slack time is detected as the number of slack cycles, which represents the difference between the worst case number of execution cycles (WCEC) of that piece of code and the number of cycles taken by its current execution (EC) divided by the current processor frequency ( $t_{slack} = \frac{WCEC-EC}{f_{CLK}}$ ). In this paper, we propose an improved, *proactive* and fully automatic method for detecting the slack time during a program execution. We rely on static analysis for discovering the correlations between parts of an application [7] and use them to partition the application in different, so-called, *scenarios*. Because we can detect the WCEC of each scenario at design time, as soon as it can be detected in which scenario an application is executed (at runtime), the processor voltage/frequency may be scaled to the adequate level. Our method is platform independent, introduces a very small runtime overhead and can be applied on top of all the existing intra-task voltage scheduling algorithms.

The paper is organized as follows. Section 2 compares our work with related approaches. A motivating example is shown in section 3. Section 4 details how scenarios may be added on top of an existing DVS-aware scheduling algorithm. In section 5, an automatic scenario-aware DVS scheduling algorithm is introduced. The experimental environment and the evaluation of our approach on two real-life benchmarks are presented in section 6. Conclusions and future plans are discussed in section 7.

## 2. RELATED WORK

An intra-task voltage scheduling mechanism which changes at runtime the supply voltage based on the splitting of a task in several slots was proposed in [11]. A similar technique was presented

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'05, September 24–27, 2005, San Francisco, California, USA.  
Copyright 2005 ACM 1-59593-149-X/05/0009 ...\$5.00.

```

for (y=0; y<3; y++)
  g(b[y]);
for (y=0; y<3; y++)
  if (ct != 1)
    f(b[y]);
  else /* ct=1 */
    g(b[y]);

```

**Figure 1: Educational example**

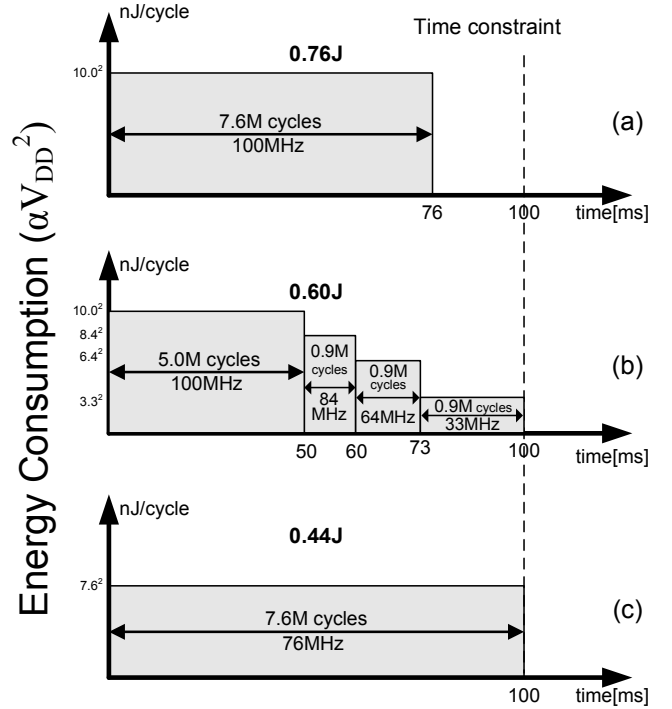
in [14] where the authors propose intelligent ways for selecting the voltage scaling points. Besides the approaches based on natural slack cycles ( $WCEC - EC$ ), in [19], Shin *et al.* propose a static method that exploits the difference between WCEC of different paths of the program. This approach has small runtime overhead and does not need any special support from the hardware or the operating system. It does not take into account the probability that a path is executed, missing some opportunities for average energy reduction. Extensions which overcome this limitation were proposed in [4, 16].

The only proactive approach that we are aware of is presented in [18]. It tries to identify the slack time in advance, using the combined data and control flow information of the program. Its disadvantages are that the data-flow analysis can not be applied easily outside of a procedure, the runtime overhead (which sometimes is big) can not be controlled, and there are no easy ways for detecting if this overhead leads to increased energy consumption. The way we select scenarios in our approach overcomes all of these limitations. As the tool and the benchmarks used for [18] are not publicly available, and the paper does not give enough information for implementing the tool, we could not directly compare our results with theirs. However, based on the same DVS-aware scheduling algorithm [19], and using real-life multimedia benchmarks, we obtain energy reductions between 14% and 52% whereas [18] reports reductions between 4% and 40%.

The scenario concept was first used in [20] to capture the data-dependent dynamic behavior inside a thread, to better schedule a multi-thread application on a heterogenous multi-processor architecture. As the authors considered also the possibility of changing the voltage level for each individual processor, their work can be considered as combining inter-task voltage scheduling with scenarios. The scenario concept was also introduced in [12], where the tasks are written using a combination of a hierarchical finite state machine (FSM) with a synchronous dataflow model (SDF). The disadvantage of this method is that the applications must be written using a limited model, which is a time expensive and error-prone operation. Currently, there are no automatic ways for translating high-level sequential programming languages (like C, which is the most used to write embedded system software) to SDF. To the best of our knowledge, in [7] we were the first to introduce a technique for automatically detecting intra-task scenarios for applications written in C. We used the discovered scenarios to reduce WCEC overestimation. In this paper, we adapt our techniques to DVS.

### 3. MOTIVATING EXAMPLE

To emphasize the possible benefit of using scenarios in intra-task DVS-aware scheduling, we start with an educational example, presented in Figure 1. Note that the function  $g$  is called three times, followed by three calls of  $f$  or  $g$ , depending on the value of  $ct$ . We assume that functions  $f$  and  $g$  do not change the value of  $ct$ . The estimated WCEC, using Shaw's timing schema [17], for this piece



(a) Only DPM, (b) DVS+DPM, (c) DVS + DPM + scenarios.

**Figure 2: An example of schedules for minimizing energy**

of code is:

$$3 \cdot (WCEC_g + \max(WCEC_f, WCEC_g)) + const,$$

where  $const$  represents the overhead of the  $if$  condition test and of the loop. Let us consider the case where

$$ct \neq 1 \text{ and } WCEC_f < WCEC_g.$$

The overestimated number of cycles in this case is  $3 \cdot (WCEC_g - WCEC_f)$ . Let us consider the numerical values

$$WCEC_f = 8 \cdot 10^5, WCEC_g = 16 \cdot 10^5, const = 4 \cdot 10^5$$

and a time constraint (deadline) of 100ms. Figure 2 (a) presents the DPM-aware voltage schedule for this case. The processor runs at a frequency (100MHz) that allows precisely meeting the timing constraint for the estimated WCEC case. As for the selected case the application execution will be finished before the deadline, the processor goes in the suspend mode. In all schedules given as examples in Figure 2, the following energy model was used: (i) for each period with a constant clock frequency  $f_{CLK}$ , the consumed energy is computed as a product of the energy consumed per cycle ( $E_{cycle} \approx V_{DD}^2$ ) and the number of cycles, (ii) the  $E_{cycle}$  in suspend mode is 0, which gives a big advantage to the schedule from (a) compared to the DVS schedules in (b) and (c), and (iii) the average time for  $V_{DD}$  switching is  $70\mu sec$ .

Figure 2 (b) shows for the same case how the DVS+DPM aware scheduler, presented in [19], works. After each evaluation of the  $if$  condition, a slack equal to  $WCEC_g - WCEC_f$  is detected; therefore, the processor voltage is reduced, still keeping the possibility of meeting the deadline.

In [7] we introduced scenarios, which are defined as *the application behavior for a specific type of input data*. The set of scenarios

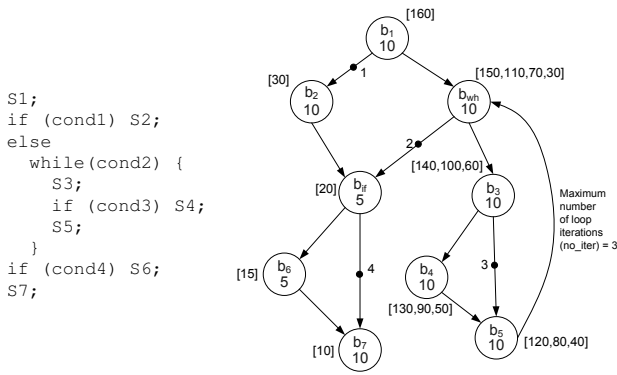


Figure 3: Another educational example

must cover all possible input data.<sup>1</sup> For each scenario, based on its WCEC, a DVS schedule is computed. All of these are combined together in the application global schedule. In the beginning of the execution, this schedule detects the current scenario and activates its schedule. There will be a little more overhead in the code than in the original DVS schedule, but our method of detecting and using scenarios, presented in section 5, keeps this overhead very low. For the example in Figure 1 two scenarios are defined, one for  $ct = 1$  and another one for  $ct \neq 1$ . Figure 2 (c) shows the voltage schedule for  $ct \neq 1$ , assuming that the scenario can be detected at the beginning of the execution and, therefore, considering as the starting voltage level the one that precisely meets the deadline given the scenario WCEC of  $3 \cdot (WCEC_f + WCEC_g) + const$ .

## 4. USING SCENARIOS IN AN INTRA-TASK DVS SCHEDULING ALGORITHM

In this section, we briefly describe a state-of-the-art intra-task voltage scheduling algorithm, introduced by Shin *et al.* in [19], and we show how the scenarios may be applied on top of it. We assume that the processor has a specific instruction `change_f_v(fCLK)`, which changes the processor frequency to  $f_{CLK}$ , adjusting the supply voltage to the corresponding voltage  $V_{DD}$ . Both  $f_{CLK}$  and  $V_{DD}$  can be set continuously within the operational range of the processor. There is a transition overhead for changing the frequency, during which the processor stops running.

### 4.1 Original DVS Scheduling Algorithm

The scheduling algorithm from [19] is based on the observation that there are large variations in the WCEC of different paths of the program. The example of Figure 3 (from [19]), which contains both a piece of code and its control flow graph (CFG), emphasizes these variations. The numbers which appear inside the CFG nodes ( $b_i$ ) represent their WCEC. The back edge from  $b_5$  to  $b_{wh}$  models the while loop, and contains its maximum number of iterations. The longest path in this example is:

$$b_1, b_{wh}, b_3, b_4, b_5, b_{wh}, b_3, b_4, b_5, b_{wh}, b_3, b_4, b_5, b_{wh}, b_{if}, b_6, b_7.$$

The WCEC of this path is 160 cycles. If the code has a deadline of  $2\mu sec$ , the processor frequency must be set to 80MHz. If, for example, the path

$$b_1, b_2, b_{if}, b_6, b_7$$

<sup>1</sup>An example of a scenario for an H.263 decoder [15] is the application behavior for any frame of type  $P$ . Together with scenarios for frame types  $I$  and  $B$ , they cover all possible behaviors.

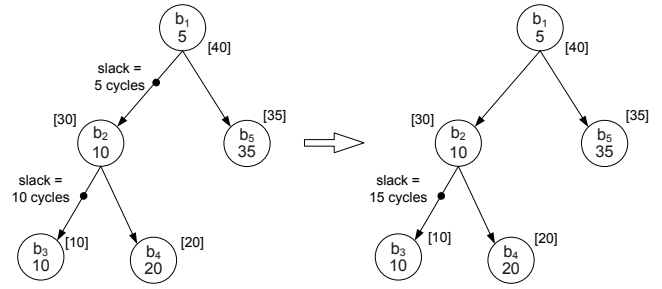


Figure 4: Slack propagation in CFG

is selected, a frequency of 20MHz is enough to meet the timing constraint.

The DVS scheduling algorithm identifies at any moment of the execution which is the longest path until its end. To do this, at compile time, for each node  $b_i$ , the remaining WCEC (RWCEC) among all the paths starting with  $b_i$  is computed. In the CFG from Figure 3, the RWCEC appears between brackets near each node. The nodes related to a loop (e.g.  $b_{wh}$ ,  $b_3$ ,  $b_4$ ,  $b_5$ ) are associated with multiple RWCEC values, one for each iteration count of the loop. Depending on the number of the loop iterations, the RWCEC table can be implemented in the scheduler as a lookup table (array) or as a formula that compute at runtime the RWCEC based on how many loop iterations were executed. The first option is more expensive from memory point of view, and the second one from computational point of view. As the aim is to reduce the energy consumed by the application, for each loop, the RWCEC implementation option that introduces the lowest energy overhead is selected.

Using the computed RWCEC, the edges  $(b_i, b_j)$  that are candidates to contain the voltage scaling points can be statically identified. In these points, code is inserted to compute the new frequency, which permits the remaining part of the application, even in the worst case, to be executed before the deadline. It also calls the `change_f_v` instruction to actually change the frequency. An edge  $(b_i, b_j)$  is a candidate if none of the longest paths starting with  $b_i$  contains it. Formally,  $(b_i, b_j)$  is selected if:

$$RWCEC_{b_i} - WCEC_{b_i} > RWCEC_{b_j} + overhead, \quad (1)$$

where *overhead* represents the cycles taken to execute the introduced code. For the loop exit nodes such as  $b_{wh}$  there are multiple options for selecting RWCEC: the largest RWCEC, the most probable RWCEC. A detailed analysis is presented in [19]. In the example of Figure 3, the selected edges are marked with a  $\bullet$ , and numbered from one to four.

As an improvement to [19], we exploit also the case when the condition from the equation 1 is evaluated to false, but

$$RWCEC_{b_i} - WCEC_{b_i} > RWCEC_{b_j}, \quad (2)$$

is true. This means that on the edge  $(b_i, b_j)$  some slack cycles appear, but they are not enough to be beneficial, in the context of DVS, for an immediate reduction of the processor supply voltage  $V_{DD}$  and frequency  $f_{CLOCK}$ . In this case, the slack cycles are propagated downwards in the application CFG, until the next candidate edge. To take into account the propagated slack cycles in edge selection, the equation 1 is modified to:

$$RWCEC_{b_i} - WCEC_{b_i} + slack_{prop} > RWCEC_{b_j} + overhead. \quad (3)$$

In figure 4, the left hand side CFG contains two selected edges:  $(b_1, b_2)$  and  $(b_2, b_3)$ . Considering a voltage scaling *overhead* larger

CFG node	Scen. 1 $cond1 = 1$	Scen. 2 $cond3 = 0$	Scen. 3 $no\_iter = 2$	Backup Scen.
$b_1$	[40]	[130]	[130]	[160]
$b_2$	[30]	[30]	[30]	[30]
$b_{wh}$	[NA]	[120, 90, 60, 30]	[120, 80, 40]	[150, 110, 70, 40]
$b_3$	[NA]	[110, 80, 50]	[110, 70]	[140, 100, 60]
$b_4$	[NA]	[NA]	[100, 60]	[130, 90, 50]
$b_5$	[NA]	[100, 70, 40]	[90, 50]	[120, 80, 40]
$b_{if}$	[20]	[20]	[20]	[20]
$b_6$	[15]	[15]	[15]	[15]
$b_7$	[10]	[10]	[10]	[10]
$VSP_1$	unused	used	used	used
$VSP_2$	unused	used	used	used
$VSP_3$	unused	unused	used	used
$VSP_4$	used	used	used	used

Table 1: RWCEC and VSPs used in each scenarios schedule

than five cycles, the right hand side CFG shows how the five slack cycles from edge  $(b_1, b_2)$  are propagated to edge  $(b_2, b_3)$ .

## 4.2 Scenarios Add-on

In section 3, a scenario was defined as the application behavior for a specific type of input data. Usually, the input data appears, sooner or later, in the application source code as values for specific variables. For example, let us assume that in the code of Figure 3, the values of variables  $cond1$  and  $cond3$  and the maximum number of while loop iterations ( $no\_iter$ ) can sometimes be directly detected based on the input data before executing  $b_1$ . Based on these values, the application can be divided in different scenarios (e.g. the header of Table 1). The backup scenario is the worst case scenario and it is used when the variable values can not be identified in advance or the overhead of adding a new scenario does not lead to (average) energy reduction<sup>2</sup>. For each scenario, the parts of the CFG that are never executed are removed and, if it is relevant, the maximum number of iterations is updated. For the remaining CFG, the RWCEC annotations and a DVS schedule are computed. Figure 5 shows the remaining CFG (the black part) for scenario 1. Table 1 presents, for each scenario, the computed RWCEC and the used voltage scaling points (VSPs) from the original DVS scenario. The VSPs that appear in a scenario schedule are a subset of the VSPs which would appear in the application schedule when scenarios were not considered. There are two reasons why a VSP may not appear in a scenario schedule: (i) its edge is not present in the scenario CFG (e.g.  $VSP_2$  and  $VSP_3$  for scenario 1) and (ii) no slack time might be discovered on its edge anymore (e.g.  $VSP_1$  for scenario 1).

To detect the runtime active scenario, at compile time *Scenario Prediction Points* (SPPs) are identified in the application. In each of them, some code to predict the current scenario, based on variable values, is inserted. The overhead introduced by this code must be small; otherwise the approach may not lead to energy reduction. Also, the earlier the current scenario is predicted, the more energy might be saved. For the previous example, one SPP is enough and it appears in the CFG on the input edge of  $b_1$ . In Figure 6 (a), it is shown as a gray node. If, for the same example, the fact that  $cond3 = 0$  can not be detected before executing  $b_1$ , but still before  $b_{wh}$ , two scenario prediction points are necessary, as shown in Figure 6 (b). The overhead introduced by this prediction code is

<sup>2</sup>If the application is executed multiple times, the scope is to reduce its average energy. For a better evaluation of the saving, the probability of execution of each scenario must be considered.

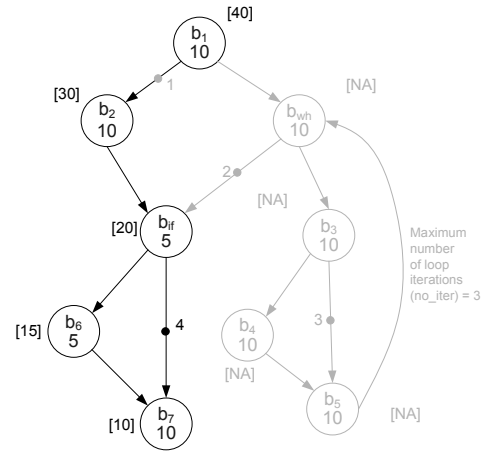


Figure 5: The CFG for scenario 1

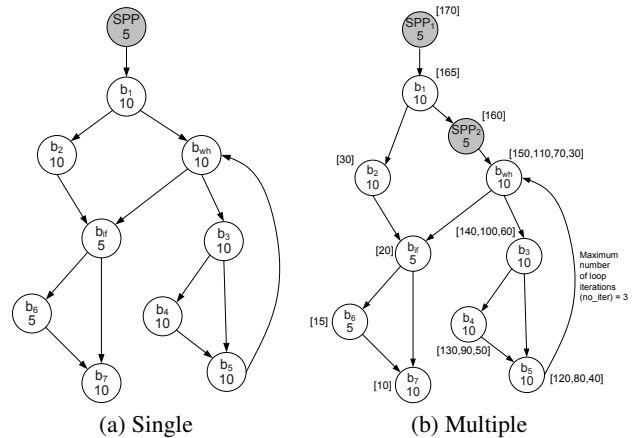


Figure 6: Scenario Prediction Points in a CFG

considered when the RWCEC is computed for the CFG nodes. (e.g. Figure 6 (b) shows the RWCEC computed for the backup scenario, considering that both  $SPP_1$  and  $SPP_2$  introduce an overhead of 5 cycles).

The scenario schedules are combined into a global schedule for the application. This schedule contains for each scenario both a list of the used voltage scaling points (VSP) and a RWCEC table with the RWCEC annotations needed in the scenario schedule (see Table 1). Besides this, it incorporates also the prediction code introduced in SPPs.

## 5. AUTOMATIC SCENARIO-AWARE DVS SCHEDULING

Our approach, based on static analysis of the application source code that is presented in section 5.1, consists of four steps: (1) identify the parameters that could potentially have an impact on the application execution time; (2) compute the maximum possible impact of these parameters on the application WCEC, using an improved version of the method from [7], adapted to the DVS requirements; (3) partition the application into possible scenarios, considering these parameters together with their impact, and select only scenarios that, in isolation, reduce the energy consumption; (4) compute a DVS schedule for each selected scenario and combine them together in the global application schedule.

$$IC_v(S) = 0 \quad (4)$$

$$IC_v(B_1; B_2) = \begin{cases} IC_v(B_2), & \text{if } v \text{ is modified in } B_2, \\ IC_v(B_1) + IC_v(B_2), & \text{otherwise.} \end{cases} \quad (5)$$

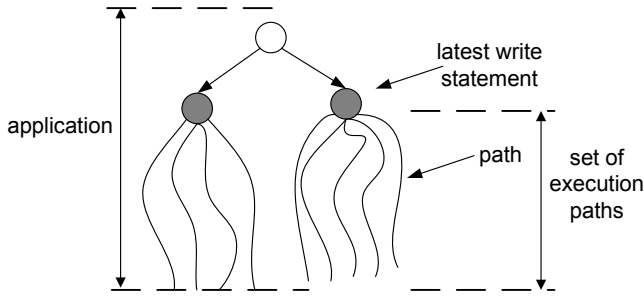
$$IC_v(\text{if } B \text{ then } B_1 \text{ else } B_2) = \begin{cases} \max(WCEC_{B_1}, WCEC_{B_2}) - \min(WCEC_{B_1} - IC_v(B_1), WCEC_{B_2} - IC_v(B_2)), & \text{if } v \text{ is compared with a constant as part of the } B \text{ condition, and } v \text{ is not modified in } B_1 \text{ and } B_2, \\ \max(IC_v(B_1), IC_v(B_2)), & \text{otherwise.} \end{cases} \quad (6)$$

$$IC_v(\text{while } B \text{ do } B_1) = \begin{cases} n_{max} \cdot WCEC_{B_1} - n_{min} \cdot (WCEC_{B_1} - IC_v(B_1)), & \text{if } v \text{ is part of the } B \text{ condition, and } v \text{ is not modified in } B_1, \\ 0, & \text{if } v \text{ is modified in } B_1, \\ n_{max} \cdot IC_v(B_1), & \text{otherwise.} \end{cases} \quad (7)$$

where  $S$  is a non-control statement,  $B, B_1, B_2$  are blocks of statements,  $n_{min}$  and  $n_{max}$  are the minimum and the maximum number of loop iterations.

(4) AST Leaf, (5) Sequential composition, (6) Conditional composition, (7) Iterative composition.

**Figure 8: The set of rules for  $IC_v$  computation.**



$$IC_v(set) = \max_{val \in values(v)} (WCEC_{set}(val)) - \min_{val \in values(v)} (WCEC_{set}(val))$$

$$IC_v(application) = \max_{set \in All\ sets} (IC_v(set))$$

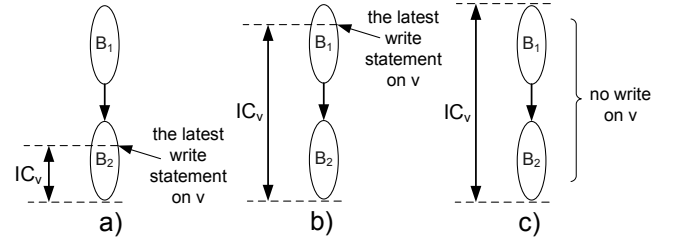
**Figure 7:  $IC_v$  Computation**

As the static analysis can not provide information about the frequency of scenario appearance, predictions can not be made about whether the energy saved due to a scenario is greater than the energy overhead introduced by it in the others. To solve this, in section 5.2, step 3 of the algorithm is augmented with a profiling based method for selecting the set of scenarios that leads the lowest energy consumption.

## 5.1 The Algorithm

The four steps of the algorithm are described below:

**1:** The first step is based on the fact that usually a few parameters have a significant impact on the application execution time (e.g. in a video decoder: image size and type). Many of these parameters are read at the beginning of the execution and remain constant for the rest of it. Moreover, there is usually only a small set of possible values for them (e.g. for an H.263 decoder, there is one variable which specifies the image type, with three possible values:  $I, B$  or  $P$ ). In a C source code, these parameters usually appear as variables or structure fields of integer or enumeration type. For each parameter, there are one or a few statements in the program that change its value (often the value is based on the program input data). In our implementation, we automatically select potential parameters



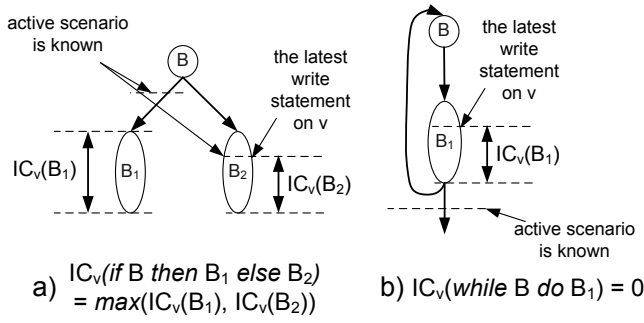
**Figure 9: IC computation for sequential composition**

based on these observations.

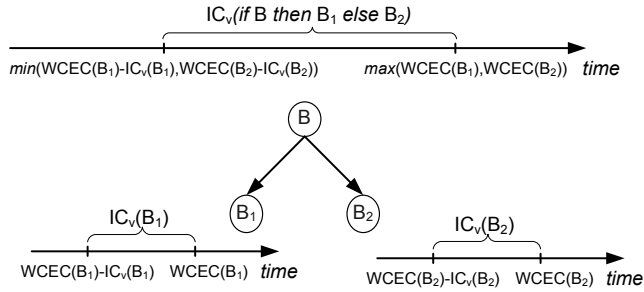
**2:** To identify which of these parameters might influence the WCEC the most, we first compute the application WCEC using Shaw's timing schema [17]. Second, the possible impact on the WCEC of each parameter (denoted by  $v$ ) is computed in the form of its so-called *influence coefficient* ( $IC$ ). The  $IC_v$  represents the maximum possible variation caused by the different values of  $v$  on the estimated application WCEC. Since it is not possible to accurately predict a scenario based on the value of  $v$  before the last write to  $v$ , we adapted the  $IC$  computation from [7] to take into account only the impact on the code after the last write statement on each execution path. Figure 7 illustrates the  $IC_v$  computation for a set of execution paths that share the latest write statement on  $v$ , and, also for an application that contains multiple such sets.

As it is not possible to enumerate all possible execution paths of a program, to compute the  $IC_v$ , a set of recursive rules are used. To this end, the abstract syntax tree (AST) of the program is traversed in a post-order manner and the  $IC_v$  is computed in each node. The AST leaves are the non-control statements of the program and the inner nodes correspond to syntactic composition of blocks of statements. Three types of composition exist: *sequential* composition, *conditional* composition and *iterative* composition. The post-order traversal of the AST allows to determine the  $IC_v$  for a program segment as a function of the  $IC_v$  values computed for its components. Each AST node type has associated one of the rules shown in Figure 8.

For a non-control statement,  $IC_v = 0$ , as there is only one possible execution path through it, meaning that there is no variation in the estimated WCEC for different values of  $v$ . For the composition nodes, if a write on  $v$  appears in their children nodes, equations



**Figure 10: IC computation for (a) conditional composition and (b) iterative composition**



**Figure 11: IC interpretation for iterative composition**

5-7 just propagate upwards the computed  $IC_v$  values. The propagation is used to compute an accurate value for  $IC_v$ , as it is used for splitting into scenarios that can only be predicted at runtime after the last write on  $v$ . Figure 9 shows how  $IC_v$  is computed for sequential composition (see equation 5) in all three possible cases: (a)  $B_2$  contains the latest write to  $v$ , (b)  $B_1$  contains the latest write to  $v$ , and (c) both  $B_1$  and  $B_2$  do not change the value of  $v$ . The last two cases are compacted in the *otherwise* part of equation 5. For conditional composition (Figure 10 (a)), if at least one of its children ( $B_1$  or  $B_2$ ) changes the value of  $v$ , during the execution of  $B$  the application active scenario is unknown. It could be discovered either after the last write from the children or on the edge between  $B$  and the child that does not modify the value of  $v$  (e.g. ( $B, B_1$ ) in the example). The  $IC_v$  computed for the loop in this case is the maximum of  $IC_v$  computed up to each point where the scenario can be determined at runtime. For iterative composition (Figure 10 (b)), if the value of  $v$  is modified in the loop body ( $B_1$ ), the application active scenario could be discovered at runtime only in the last iteration of the loop. As it is almost impossible for most of the loops to indicate in advance which is the last iteration, the active scenario is discovered only after the loop and hence, in this case, computed  $IC_v = 0$ .

Only if  $v$  is part of an *if* or *while* condition, then the estimated WCEC for the associated composition node may vary based on the value of  $v$ . If  $v$  is not part of the condition, its value does not influence which *if* branch is taken or how many *loop* iterations will be executed. Therefore, equations 6 and 7 are the only ones that inject values different from 0 in the recursive computation of  $IC_v$ . Figure 11 graphically interprets how  $IC_v$  is computed, in the equation 6, as the difference between the WCEC of the longest possible execution path (max term) and the WCEC of the shortest one (min term having as arguments the WCEC of the blocks minus the impact of these blocks). As later on, for splitting into scenarios, only

the comparisons of variables with constants are considered<sup>3</sup>, this is taken into account in equation 6. For iterative composition, two distinct cases appear when the loop body does not change the value of  $v$ : when  $v$  is not part of the condition, and when it is (equation 7). The first case is a natural extension of the sequential composition, where the nodes  $B$  and  $B_1$  are executed for  $n_{max}$  times. When  $v$  is part of the loop condition, the  $IC_v$  is computed as the difference between the lengths of the longest possible execution path through the loop (the term that contains  $n_{max}$ ) and of the shortest one (the one with  $n_{min}$ ).

To go beyond function borders, for each function call that has  $v$  as a parameter, a renaming is done for computing the  $IC_v$  inside the function.

The four types of AST nodes cover the entire ANSI C syntax. Simple control flow statements, like *for*, *switch*, *goto*, can be directly mapped into *while* and *if* statements without affecting their WCEC. A few constructs are difficult to handle: recursive functions (unknown depth), back jumps (hidden loops) and dynamic function calls. The first two can be transformed in loops using different mechanisms [5]. Even though the dynamic function call seems to be a fundamental problem, it is solvable in embedded software as, usually, all possible called functions are known at design time.

In the end, the root of the AST yields the values of the  $IC$ s computed for each possible parameter.

**3:** To avoid an explosion in the number of scenarios, different criteria for selecting parameters to define scenarios might be used. The selection may incorporate knowledge about the application combined with heuristics based on the computed values of  $IC$ s. An example of a very simple heuristic is to select only those parameters with very big  $IC$  values.

For each *selected* parameter, the constants it is compared to in the source code are collected. These constants, together with the comparison operators, are used to split the set of possible values of the parameter into subsets. A scenario is characterized in the end, by the possible values of the selected parameters.

Figure 12 shows how the  $IC$  for the variable  $ct$  is computed for the example presented in section 3. As it could already be seen in the source code, we can automatically detect two scenarios based on the values of  $ct$ : one corresponding to  $ct = 1$  and the other to  $ct \neq 1$ . The splitting into scenarios does not depend on the variable  $y$  as  $IC_y = 0$  (because  $y$  changes its value in both *for* loops).

For each potential scenario, by using static analysis, it is computed whether, considering the overhead for scenario detection and scheduling, energy is saved when it is run. This overhead influences the energy consumption in two ways: (i) the prediction code increases the number of execution cycles and the code size (more instruction memory involves more energy) and (ii) the sizes of the RWCEC tables used by the global schedule increase. There is no supplementary cycle overhead for processor frequency computation and changing when compared to traditional DVS scheduling, as no new voltage scaling points are added in the program. If a potential scenario is not energy beneficial, it will be merged with another one.

**4:** For each scenario, a DVS-aware schedule is computed (e.g. using the method from [19]). All of those schedules are combined into a global one, as presented in section 4.2. This schedule also includes code for predicting the active scenario. For each of the parameters used for splitting into scenarios, this kind of code is inserted in the points which are for sure not followed by a statement that changes the parameter value. It consists of the variable

<sup>3</sup>The limitation appears as the scenario selection algorithm is applied at design time.

source code	$IC_{ct}$ equation	$IC_{ct}$ value
1 for (y=0; y<3; y++)	$3 \cdot IC_{ct}(g) + 3 \cdot [\max(WCEC_f, WCEC_g) - \min(WCEC_f - IC_{ct}(f), WCEC_g - IC_{ct}(g))]$	$24 \cdot 10^5$
2 g(b[y]);	$IC_{ct}(g)$	0
3 for (y=0; y<3; y++)	$3 \cdot [\max(WCEC_f, WCEC_g) - \min(WCEC_f - IC_{ct}(f), WCEC_g - IC_{ct}(g))]$	$24 \cdot 10^5$
4 if (ct != 1)	$\max(WCEC_f, WCEC_g) - \min(WCEC_f - IC_{ct}(f), WCEC_g - IC_{ct}(g))$	$8 \cdot 10^5$
5 f(b[y]);	$IC_{ct}(f)$	0
6 else /* ct=1 */		
7 g(b[y]);	$IC_{ct}(g)$	0

Numerical values:  $WCEC_f = 8 \cdot 10^5$ ,  $WCEC_g = 16 \cdot 10^5$ ,  $IC_{ct}(f) = 0$ ,  $IC_{ct}(g) = 0$

Figure 12:  $IC_{ct}$  computation for the educational example from Figure 1

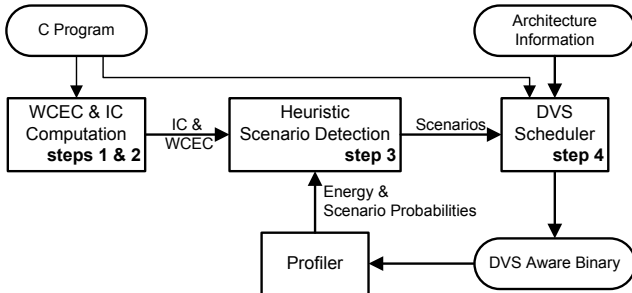


Figure 13: Scenario-aware DVS scheduling work-flow.

comparisons also used for the splitting.

## 5.2 Profiling for Average Energy Reduction

A scenario, generated in step 3 of our algorithm, is always beneficial for energy when it is selected at runtime. However, it causes an overhead if it is not active. If the scenario does not appear frequently enough at runtime, the total energy saved by it might be less than the energy consumed by the overhead introduced by it in the other scenarios. The following inequality is used to detect the impact of a scenario  $S$ , with a probability of appearance  $p(S) \in [0, 1]$ , on the average energy consumption of the application:

$$E_{saved}(S) * p(S) > E_{overhead}(S) * (1 - p(S))$$

The static analysis can not detect if the average energy of the application increases or decreases when a scenario is introduced. To gather the necessary information, in Figure 13, it is shown how the original algorithm is augmented with a profiling step. It provides energy information per scenario and scenario frequencies to the heuristic from step 3 of the above presented algorithm. This approach allows multiple iterations over steps 3 and 4 of the algorithm, which may lead to progressive refinement of energy improvement.

## 6. EXPERIMENTAL EVALUATION

All the presented steps were implemented on top of SUIF [2]. For our experiments, we used a micro-architecture model similar to ARM7TDMI [1]. We consider that both  $f_{CLK}$  and  $V_{DD}$  can be set continuously within the operational range of the processor and that there is a transition overhead of  $70\mu sec$  for changing the frequency, during which the processor stops running. The considered overhead is quite large. For computing the WCEC of scenarios, we use Shaw's timing schema [17].

We tested our method on two multimedia applications, an MP3 decoder [10] and a restricted H.263 decoder [15] that supports only  $I$  and  $P$  frames. We show that applying our approach on top of the DVS-aware scheduling algorithm from [19] results in a significant energy reduction.

Variable Name	IC
no_channels	1.202.984
b_type	1.053.728
mode_ext	104.418
mixed_flag	66.128

Table 2: Variables' influence coefficients for MP3 Decoder

For the MP3 decoder, we computed the influence coefficient ( $IC$ ) for all possible parameters. The ones with  $IC$  value bigger than 100 are listed in Table 2. As there is a big difference between the influence coefficient values of the second and the third parameter, initially, we consider only the first two parameters for splitting into scenarios. The first line of Table 3 shows both the average energy improvement and the overhead introduced in the application by using these scenarios (cycles, instruction memory, and data memory for the RWCEC tables). Both energy reduction and overhead are based on the DVS algorithm of [19] as a reference. We have chosen two sets of input files, one taken from [6] and one consisting of randomly selected stereo songs downloaded from the internet. The first set, even if it is a standard benchmark for mp3 decoders, is not representative in our opinion because it tests all the extreme coding cases (which rarely appear in usual mp3 files) and the mono songs (which are not often listened to). By considering in step 3 of our algorithm the other two parameters as well, and based on profiling information, we obtain progressively the sets of scenarios shown in the rest of the table. The best splitting is the one shown in the second line, with leads to an energy reduction of almost 16%<sup>4</sup>. The experiments show that including *mixed\_flag* in the scenario definition introduces more overhead than savings because the extra scenarios occur too infrequently.

For the H.263 decoder, the set of scenarios that reduces the energy consumption the most has one scenario for  $I$  frames and one scenario for  $P$  frames. As the processing performed for an  $I$  frame is a true subset of the processing done for a  $P$  frame, the application WCEC is equal with the WCEC of the scenario for  $P$  frames, which is also the backup scenario. Therefore, the only scenario that reduces the energy consumption is the one for  $I$  frames. Depending on the input stream structure, we obtained an energy reduction from 14% (for an input stream which contains for each  $I$  frame six  $P$  frames) to 45% (if the input stream contains an equal number of  $I$  and  $P$  frames).

## 7. CONCLUSIONS AND FUTURE WORK

We have presented an automatic scenario-aware DVS scheduling algorithm for reducing the energy consumption of real-time applications. It can be applied on top of all intra-task DVS-aware scheduling techniques, making them more effective. It is based on

<sup>4</sup>For the input files from [6], the highest average energy reduction, of 52%, is given by the fourth set of scenarios.

Scenarios Set	Number of Scenarios	Energy Reduction			Overhead		Used Variables
		Random Stereo	Benchmark from [6]	WCEC (cycles)	I memory (bytes)	D memory (bytes)	
Set1	4	14.98%	51.66%	46	139	1108	no_channels, b_type
Set2	10	<b>15.90%</b>	51.90%	104	235	2424	no_channels, b_type, mode_ext
Set3	6	14.98%	51.81%	74	349	3452	no_channels, b_type, mixed_flag
Set4	15	15.88%	<b>52.03%</b>	188	661	6816	no_channels, b_type, mode_ext, mixed_flag

**Table 3: Energy reduction for MP3 Decoder**

scenarios that incorporate both the correlations between different parts of the application source code and different numbers of iterations for loops. To discover scenarios, we propose an algorithm based on static analysis augmented with profiling information. This algorithm guarantees a small runtime overhead for scenario prediction, and determines at design time which is the set of scenarios that yields the largest energy reduction. Our method was tested on two applications and an energy reduction between 14% and 52% was obtained when compared with traditional DVS scheduling.

There is a limitation in how close to the beginning of a program a scenario can be detected, and the processor supply and clock frequency can be adapted for it. Due to this, there is also a limitation in the reduction of the energy consumption as, before a scenario is detected, the worst case situation is taken into account. We plan to surpass this limitation by considering a probabilistic approach for predicting in advance on the basis of partial information in which scenario the application will end up. We also want to study a combination of intra and inter-task scenario based voltage scaling for multiprocessors systems.

## 8. REFERENCES

- [1] ARM7TDMI datasheet. <http://www.arm.com/products/CPUs/ARM7TDMI.html>.
- [2] S. Amarasinghe, J. Anderson, M. Lam, and C. W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proc. of the 7th Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995.
- [3] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. A. Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *Proc. of the IEEE Design, Automation and Test in Europe Conference*, pages 514 – 519, Munich, Germany, March 2005.
- [4] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proc. of the IEEE Design, Automation and Test in Europe Conference*, pages 168 – 175, Paris, France, March 2002.
- [5] J. Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, 171(2):156–182, Dec. 2001.
- [6] M. D. et al. MPEG-1 audio layer III test bitstream package, May 1994.
- [7] V. S. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved wcet estimation. In *Proc. of the 42nd ACM Design Automation Conference*, pages 101–104, Anaheim, CA, USA, June 2005.
- [8] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of the International Symposium on Low Power Electronics and Design*, pages 197 – 202, August 1998.
- [9] N. K. Jha. Low power system scheduling and synthesis. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 259–263, San Jose, CA, USA, November 2001.
- [10] K. Lagerström. Design and implementation of an MP3 decoder, May 2001. M.Sc. thesis, Chalmers University of Technology, Sweden.
- [11] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proc. of the 37th Design Automation Conference*, pages 806 – 809, Los Angeles, CA, USA, June 2000.
- [12] S. Lee, S. Yoo, and K. Choi. An intra-task dynamic voltage scaling method for SoC design with hierarchical FSM and synchronous dataflow model. In *Proc. of the International Symposium on Low Power Electronics and Design*, pages 84–87, Monterey, CA, USA, August 2002.
- [13] Y.-H. Lu, L. Benini, and G. D. Micheli. Low power task scheduling for multiple devices. In *Proc. of the International Workshop in Hardware/Software Co-design*, pages 39–45, March 2000.
- [14] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Proc. of the Workshop on Compilers and Operating Systems for Low Power*, October 2000.
- [15] K. Rijkse. Video coding for narrow telecommunication channels at <64kbits/s. Technical report, Telenor R&D, 1995.
- [16] J. Seo, T. Kim, and K.-S. Chung. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proc. of the 41st Design Automation Conference*, pages 87–92, San Diego, CA, USA, June 2004.
- [17] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
- [18] D. Shin and J. Kim. Optimizing intra-task voltage scheduling using data flow analysis. In *Proc. of the 10th Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.
- [19] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy, hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, March 2001.
- [20] P. Yang, et al. *Multi-Processor Systems on Chip*, chapter Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems. Morgan Kaufmann, 2003.
- [21] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 84 – 93, May 2004.