Run-Time Precomputation of Data-Dependent Parameters in Embedded Systems

ELENA HAMMARI and PER GUNNAR KJELDSBERG, Norwegian University of Science and Technology, Norway
FRANCKY CATTHOOR, IMEC, Belgium and University of Leuven, Belgium

In many modern embedded systems the available resources, e.g., CPU clock cycles, memory and energy, are consumed non-uniformly while the system is under exploitation. Typically, the resource requirements in the system change with different input data that the system process. This data trigger different parts of the embedded software resulting in different operations executed that require different hardware platform resources to be used. A significant research effort has been dedicated to develop mechanisms for run-time resource management, e.g. branch prediction for pipelined processors, prefetching of data from main memory to cache, and scenario based design methodologies. All these techniques rely on the availability of information at run-time about the upcoming changes in the resource requirements. In this paper we propose a method for detection of upcoming resource changes based on preliminary calculation of software variables that have the most dynamic impact on the resource requirements in the system. We apply the method on a modified real-life biomedical algorithm with real input data and estimate a 40% energy reduction as compared to static DVFS scheduling. Comparing to dynamic dvfs scheduling, an 18% energy reduction is demonstrated.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; **Firmware**; *Real-time systems*; • **Hardware** → On-chip resource management;

Additional Key Words and Phrases: dynamic embedded system, run-time resource management, parameter precomputation

ACM Reference Format:
Elena Hammari, Per Gunnar Kjeldsberg, and Francky Catthoor. 2018. Run-Time Precomputation of Data-Dependent Parameters in Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 1, 1 (March 2018), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern embedded systems are designed with stringent resource constraints and growing complexity. To meet the design goals advanced design methodologies are needed to keep the resource usage under control. This becomes particularly challenging when dynamic run-time behavior renders standard static design time optimization, like static schedulers and timing analysis, insufficient. The designer can then either use a worst case approach, which will typically result in resource waste and extended computation time, or try to estimate upcoming system requirements to schedule tasks and resources in a more optimized way.

Authors' addresses: Elena Hammari, Elena.Hammari@ntnu.no; Per Gunnar Kjeldsberg, Per.Gunnar.Kjeldsberg@ntnu.no, Norwegian University of Science and Technology, Department of Electronic Systems, O.S. Bragstads plass 2b, Trondheim, NO-7491, Norway; Francky Catthoor, IMEC, Kapeldreef 75, Leuven, 3001, Belgium, University of Leuven, Department of Electrical Engineering, Leuven, Belgium, Francky.Catthoor@imec.be.

Branch prediction is an example of a technique that tries to exploit the resources of a pipelined processor architecture as effectively as possible. Speculatively following one of the branches in an if-then-else structure, it is possible to continue pipeline execution even if the required branch test result is still not available from later pipeline stages. If it turns out that the branch selection is wrong, the speculatively executed instructions are discarded (wasted) and the pipeline restarts along the correct branch, introducing an added delay. The branch prediction can be done in several ways, e.g., through use of statistic knowledge about the application or recent dynamic behavior collected in a history table. In this case, estimation of upcoming system requirements is based on prediction. A prediction is inherently inaccurate and incurs a fluctuating waste of time and resources that is unknown to the scheduler in advance.

Another situation where prediction of future system behavior is applied is in data prefetching. Data is moved from background memory to cache based on an assumption that it will be needed in the near future. Depending on the complexity of the access patterns, it is easy or difficult to predict the upcoming data needs. Incorrect predictions incur unnecessary data transfers, costly both in time and energy, and may even end up evicting useful data from the cache reducing the hit ratio.

As part of the system software, scenarios at different abstraction levels can be used for efficient scheduling and resource utilization. Traditional use-case scenarios and work load scenarios are extracted by analyzing the different usage episodes of the system in terms of user actions and the system operations. The system is adapted to perform required functionality of the upcoming working situation. Compared with these high-level scenarios that do not take actual resource requirements into account when adapting the platform, the system scenarios design methodology allows run-time management of system resources based on a set of system scenarios specified at design time [8]. System scenarios group application execution patterns with similar resource requirements. These patterns are found by profiling the application on the target platform and observing the variables in the application code having the most impact on the resource costs. At run-time, these variables, that we will call parameters, are monitored and used to identify the active scenario, adapting the platform if a scenario switch is needed. The existing approaches for estimation of parameter values are either based on statistical methods [8] that can lead to missed deadlines or assume that parameter values are known at the beginning of the task graph [8] [10], which is only applicable to a limited number of applications.

Another field that can benefit from upfront estimation of resource requirements is energy-aware online scheduling for real-time embedded systems [1]. These scheduling algorithms perform on-the-fly scheduling of application tasks that guarantees that the application finishes within a deadline and at the same time dynamically control system's power management mechanisms to reduce power consumption. Dynamic Voltage and Frequency Scaling (DVFS) is a widely used power management mechanism in many modern embedded systems. DVFS-based online scheduling algorithms exploit dynamic slack, which represents the unused time that becomes available when tasks complete earlier than the expected worst case execution time. If the information about the execution time of tasks (i.e. future dynamic slack) is available upfront, the algorithms can adjust power earlier and reduce number of switches. However, to be applicable in hard real-time systems, where no deadline misses are allowed, the estimation cannot use probabilistic approach. One idea could be to monitor application variables with most impact on the execution time of the tasks, i.e. parameters, but in the general case their values are not known before the task starts.

In this paper we propose a technique for precomputation of data-dependent parameters at run-time. A precomputation code is added to the application at an early point and provides high accuracy upfront information on changing resources requirements to run-time scheduler. The technique is applicable in hard-real time systems, but requires that the system supports communication

ALGORITHM 1: Example code

```
Input: $d$, dynamic data
...
while $i < 1000$ do
    if $e(i, d)$ then
        $A$ ; // 100 clock cycles
    end
    else
        $B$ ; // 20 clock cycles
    end
end
```

between the application and the run-time scheduler. It also incurs an overhead that depends on the application and the input data, but we show that for a real practical case the gains from early and accurate detection of upcoming resource requirements outweighs the precomputation overhead.

The paper is organized as follows. Section 2 demonstrates the motivation for this work. The related research is reviewed in Section 3. Section 4 describes in detail the proposed method. The experimental results are presented in Section 5 and discussed in Section 6. Finally, Section 7 concludes the paper.

2 MOTIVATIONAL EXAMPLE

Consider a small example code in Algorithm 1. The application contains a loop repeating 1000 times including a conditional statement $e(i, d)$ that depends on an input value d.When $e(i, d)$ is true, code block A is executed, which takes 100 clock cycles. Otherwise, code block B is executed taking 20 clock cycles. Let s be the total number of clock cycles it takes to evaluate the if-else clause and the $e(i, d)$ statement. The check required for one execution of the while-loop is also included in s. We assume that the input value d is such that A executes 50% of the time and B executes the remaining 50% of the time. Total execution time (t_{ET}) to run the loop is calculated to be: $t_{ET} = 1000 \cdot s + 500 \cdot 100 + 500 \cdot 20 = 1000 \cdot s + 60000$ clock cycles.

Since the condition depends on the input, which is not known before the execution of the code, the system scheduler must assume the worst case where the block A runs all 1000 times. The worst case execution time (t_{WCET}) is then given by: $t_{WCET} = 1000 \cdot s + 100000$ clock cycles. Thus, the system scheduler will assign t_{WCET} to this job resulting in 40000 clock cycles overestimation compared to the actual execution. The classical scheduling and timing analysis tools use this approach to estimate required clock cycles, where only static code analysis is performed. Modern run-time scheduling algorithms make use of execution time slacks: when a task has finished before its deadline, the slack is used to schedule an earlier execution of the following tasks. This avoids the waste of clock cycles, but the overestimation hinders optimization of energy consumption. Many modern embedded processors implement several power-frequency modes (DVFS) that can be used at run-time to reduce energy in case of lower load. Accurate upfront estimation of required clock cycles is crucial for the efficiency of DVFS-based methods. An overestimation means that the processor cannot be switched to low-power mode before the execution completes, so less energy is saved. An underestimation means that the processor is switched to a slower mode than required which will cause deadline misses, not acceptable in hard real-time systems.

The solution to this problem can be to precalculate the required clock cycles before the data-dependent loop. If we copy the loop structure with the condition and place it as early as possible before the loop starts as shown in Algorithm 2, the number of times each branch runs in the loop

ALGORITHM 2: Example code with precomputation

```
Input: $d$, dynamic data
// Precomputing $c$ - number of runs of code block A:
int $c = 0$ while $i < 1000$ do
    if $e(i, d)$ then
        $c$++ ; // 1 clock cycle
    end
end
// Original data-dependent loop:
while $i < 1000$ do
    if $e(i, d)$ then
        $A$ ; // 100 clock cycles
    end
    else
        $B$ ; // 20 clock cycles
    end
end
```

can be calculated in advance based on the available input value d. The scheduler should allocate for this precomputation $1000 \cdot s + 1000$ clock cycles, so the total execution time with precomputation (t_{PET}) is given by: $t_{PET} = t_{ET} + 1000 \cdot s + 1000 = 2000 \cdot s + 61000$. Each increment of c takes 1 clock cycle, in total a maximum of 1000 clock cycles. Right after the precomputation completes, the exact number of cycles expected to be used in the loop (t_{ET}) is known and an evaluation can be done whether rescheduling will save clock cycles. This method reduces overestimation by $40000 - 1000 \cdot s - 1000 = 39000 - 1000 \cdot s$ clock cycles so the processor can run on a lower voltage while the task still meet its deadline. Note that this gain depends on the time spent on the evaluation of the condition. If s is close to the number of clock cycles it takes to execute either of the two branches (here if $s = 29$), there will be no actual gain of using precomputation. On the other hand, in many cases, the precalculation can be done as an integrated part of the earlier code sections, e.g. when the data value d is generated. This can partially or completely remove the precomputation overhead.

In reality, real-life applications can have good potential to apply precomputation as we will show in Section 4. With the exact knowledge of the number of clock cycles required for a given data set, the scheduler can improve the resource utilization, e.g., through dynamic voltage and frequency scaling to save energy. Similarly, precomputed knowledge of correct branch selection or future data needs can increase a system's pipeline and prefetching efficiency.

3 RELATED WORK

As outlined in Section 1, estimation of upcoming system requirements is important in many situations, e.g., for scheduling and allocation in scenario based settings, for branch prediction and data prefetching. In this section we will present an overview of state of the art estimation techniques applied in these and similar settings. Most of the estimation techniques use statistical prediction methods. All such methods have a certain percentage of mispredictions leading to data-dependent and therefore non-deterministic overheads. Such overheads present a problem for energy optimization in hard real time embedded systems.

Branch prediction is an essential part of pipelined execution. The prediction can be as simple as statically always taking the branch following immediately in the next sequential instruction (i.e., not perform a jump). It is then the task of the compiler to place the most likely branch at this position [11]. The simplest dynamic predictor uses a 2-bit (or even 1-bit) buffer to keep track of the most likely branch to be taken, based on the recent history of actual branch selection [17]. Since there can be very many branches in an application, a cache-like structure using the lower instruction address bits as index is typically used in the history buffer. To improve the prediction accuracy it is possible to use counters with more bits and buffers with increased size. Two-level predictors are also used since there may be a pattern in the branch selection. A branch history shift register is used to track the recent branch selection, and the current value is used to choose between different counters in a pattern history table, which value is then used for the prediction [23]. In addition to have increasing resources and time overhead, the prediction accuracy is very application dependent. For applications where 100 % accuracy can be achieved through precomputation, the technique presented in this paper is a favorable approach.

Data prefetching is used in cache based memory systems for increased hit ratio. For example, a sequential prefetcher simply fetches the next cache block from background memory when a given cache block is accessed [20]. The prediction is hence that data from the next cache block will be needed in the near future. If this is not the case, it can be seen as a misprediction, and an overhead is introduced in the unnecessary data transfer. More advanced schemes like Stride Directed Prefetching avoids needless prefetches when the processor's referencing pattern strides through nonconsecutive memory blocks [5]. Further prediction enhancements include use of Reference Prediction Tables [2], Global History Buffers [18], and Delta Correlation Prediction Tables [9] to dynamically update stride distances based on encountered cache misses. The overhead, both in large history tables as well as time and energy consuming data transfers due to misprediction, makes precomputation attractive for relevant applications.

The system-scenario-based design methodology has been developed to improve resource utilization in embedded systems with data-dependent resource requirements. In [8] the authors provide an overview of the methodology and its early use. Recent published reports provide extensions on memory aware system scenarios [4], power aware system scenarios for wireless transmitters [24], and scenario aware synchronous data flow graphs [19]. System scenarios utilize the correlation between the resource requirements of an application and values of the variables in the application software, which we call parameters. At run-time, a system scenario is identified from the actual parameter values and a resource management mechanism is invoked to adapt to the changing resource requirements. Scenario identification and switching overheads must be minimized. Two approaches for prediction of scenario switch exist in this framework. The first one [8] [10] assumes that parameters are available at the beginning of the task graph before the scenario starts. Then depending on the definition space of the parameters, either a multi-valued decision diagram [8] or polyhedral subdivision of the parameter definition space [10] is selected for scenario identification. However in many real-life applications the data-dependent parameter is not known before the scenario starts, they must be estimated beforehand. Two examples are the short-term Lyapunov exponent computation (STLEC), which will be studied in this paper, and the fundamental frequency detector (FFD), where the execution time varies based on the number of peaks in the autocorrelation function of the input signal. The aforementioned methods for scenario identification cannot be used directly in these cases. The second approach [8] is to predict the values of the parameters using a probabilistic approach, which can have mispredictions that may cause deadline misses. Thus, this approach is not relevant for hard real-time systems. Precomputation of data-dependent parameters proposed in this paper enables 100% accurate scenario detection at an

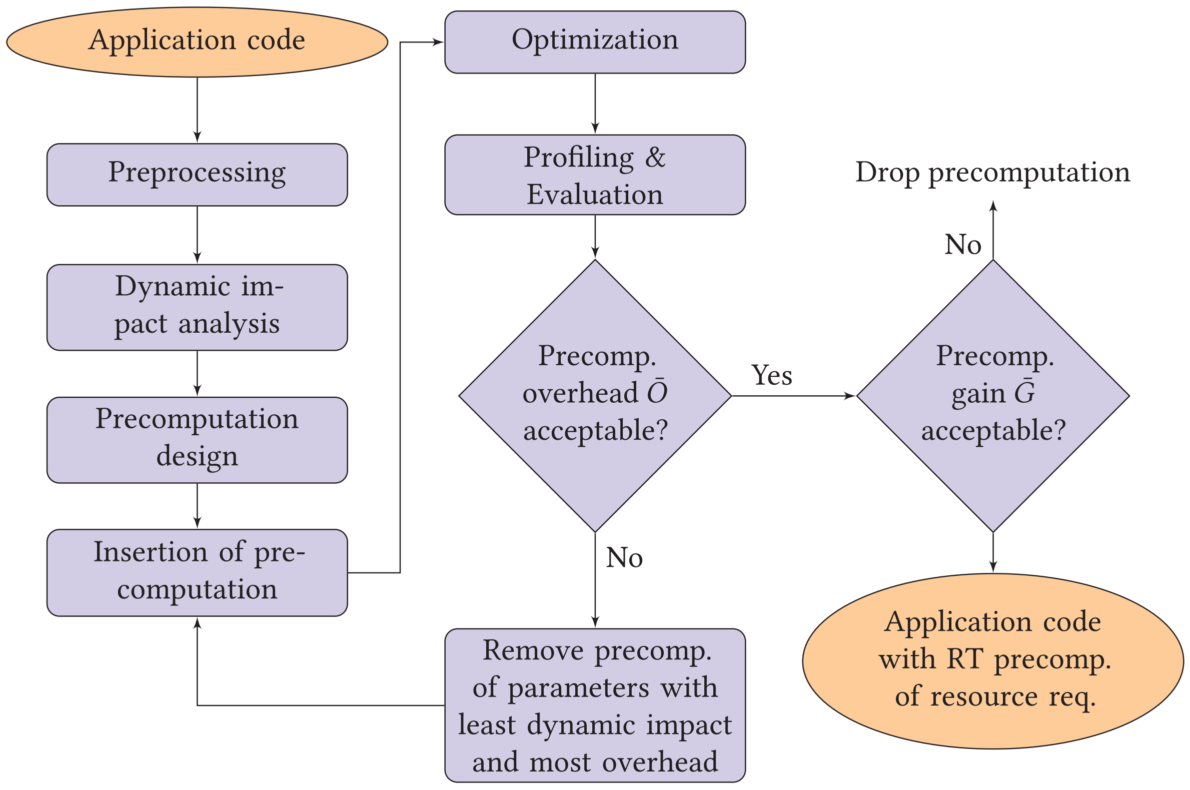
Application code
Preprocessing
Dynamic impact analysis
Precomputation design
Insertion of precomputation
Optimization
Profiling & Evaluation
Precomp. overhead $\bar{O}$ acceptable?
Yes
No
Remove precomp. of parameters with least dynamic impact and most overhead
Precomp. gain $\bar{G}$ acceptable?
No
Drop precomputation
Application code with RT precomp. of resource req.

Fig. 1. Precomputation of run-time resource requirements

earlier point in the application and thus an earlier invocation of run-time resource management mechanisms

DVFS-based online real-time schedulers are used in real-time embedded systems running applications with data-dependent execution times. These algorithms optimize system's power consumption by dynamically adjusting voltage and frequency of the processor and schedule the application tasks to finish within a deadline [1]. They initially assume that each task will take worst case number of cycles to execute and select supply voltage/processor speed accordingly. When a task is completed, they calculate the available time slack and use it to reschedule the consecutive tasks and update the voltage/speed setting. This scheduling can be improved by providing the scheduler upfront information on the execution time of tasks, for example by estimating values of software variables with the most impact on the execution time (parameters). This paper introduces a technique for run-time precomputation of such data-dependent parameters and investigates the case of using this technique together with a DVFS-based online real-time scheduler in a real embedded system. To the best of our knowledge, only probabilistic approach for prediction of execution time of tasks have been tried earlier, which cannot be directly applied to hard-real time embedded systems.

4 DESCRIPTION OF PROPOSED METHOD

The proposed method for run-time estimation of resource requirements consists of several steps shown in Figure 1. The method is aimed at dynamic embedded applications, i.e. applications having varying resource requirements throughout their execution on the target embedded platform and where the resource requirement variation is caused by changing values of data inputs. Let d_k denote a value of input k. Given n number of inputs, we specify a vector $\mathbf{d} = d_1, ...d_n$ containing values of all data inputs. The idea behind the method is to add a small precomputation code early

ALGORITHM 3: Representative dynamic code structure

```
Input: $d$, dynamic data
...
Initialize loop iterators $i, j$
while $e_0(i,d)$ do
    $i = A(i)$;
    if $e_1(i,d)$ then                       // Single or grouped evaluation expression
        $B$;
    end
    else if $e_2(i,d)$ then                  // Nested conditional statement
        $C$;
        if $e_3(i,d)$ then
            $D$;
        end
        else
            $E$;
        end
    end
    else if $e_4(i,d)$ then
        $F$;
        continue/break;                      // Loop control
    end
    else
        $G$;
    end
    while $e_5(i,j,d)$ do                    // Nested loop
        $j = H(j)$;
        $I$;
    end
end
```

in the application.Based on the available run-time input data values **d**, this code will calculate in advance the resource requirements of the application parts having the most significant dynamic impact on the overall resource requirements of the application. This information can be then exploited by run-time resource management mechanisms to dynamically and efficiently reassign system resources to the application according to its run-time resource requirements.

Different resource requirements may be generally considered, e.g. CPU time, memory accesses, clocking and power gating of different parts of the platform. We will here demonstrate the method for CPU time resource requirements only. We will also target real-time embedded systems with critical timing deadlines. Such systems require all components to be real-time predictable, so we do not consider effects of cache and preemptive scheduling. The assumptions for the precomputation to work are:

(1) Resource requirements (CPU time) of the application vary with input data and depend on values of internal software variables (data-dependent parameters).
(2) A resource management mechanism (run-time scheduler) is present that can benefit from precomputation.
(3) Input data is only read at the beginning of a task graph.

(4) The only changes to variables come from the calculations in the tasks.
(5) Data-dependent parameters remain stable throughout the dynamic parts of the code that they control

Assumptions 3-5 ensure that it is always possible to precompute a variable by simply duplicating all statements that update it to an earlier position in the code. Reading data at the beginning of a task graph is very common for embedded real-time applications that operate on data frames/packages, f.ex. video decoders, STLEC, image compression etc. Further research is needed to investigate other types of applications and possibly relax the assumptions.

In each step of the method we will first present general equations for n data inputs, where it is possible, and then illustrate its use on a simple case with a single data input. Algorithm 3 presents the simple case. The example code contains a nested loop with multiple branches and code blocks that perform some data processing. The following control structures, present in most embedded applications [3], are considered:

(1) Loop
(2) Branch
(3) Nested loop
(4) Nested branch
(5) Loop exit statement
(6) A combination of these control structures

We assume that this is a dynamic application, i.e. the execution time of the application varies over time and that the variation is caused by changing values of a data input. Control structures have the most impact on the execution time variation. The code includes multiple evaluation expressions e_0,...,e_5 that depend on a data input with value d. For different values d, the code blocks A,...,I will be executed a different number of times. Observing that the execution times of the code blocks are different, the time it takes to execute the given code on an embedded platform varies with d. Let us assume that the variation of the required CPU time is significant while the available CPU time is limited, e.g., due to application deadlines and other competing processes. The system contains a run-time scheduler, which does the resource (CPU time) management. By following the steps in Figure 1, we will design a mechanism that will provide the scheduler a run-time estimate for the amount of CPU time the application will require the next time it executes the loop in Algorithm 3.

4.1 Preprocess the application

In this step the application source code is reorganized to bring it into the form that can be efficiently processed by the subsequent step of the dynamic impact analysis. In this process the code is divided into code blocks with nearly static resource requirements and code structures having a dynamic impact on the resource requirements.

In our demonstration case the resource to be optimized is the CPU time and the Algorithm 3 represents the output of the preprocessing step. Code blocks A,...,I have execution times $t_A(d)$,...,$t_I(d)$, and evaluation expressions e_0,...,e_5 have execution times $t_{e_0}(d)$,...,$t_{e_5}(d)$. These execution times are nearly static, i.e, they have no or only a small dependency on d. For a code block/evaluation expression X, having execution time t_X, the difference δ_X between its upper and lower execution time is constrained by the maximum allowed execution time difference δ as given by Equation 1.

$$\delta_X \leq \delta \text{ for all } X, \text{ where } \delta_X = \max\left(t_X\left(d\right)\right) - \min\left(t_X\left(d\right)\right) \tag{1}$$

The maximum allowed execution time difference δ defines the borders of code blocks/evaluation expressions and therefore also the size of the code hidden in them and to what extent the control code structures depending on d are revealed. The lower δ is selected, the more and smaller code blocks there will be and the more dynamic code structures will be revealed. And vice versa, the higher δ, the fewer and larger code blocks there will be and less revealed dynamic code structures. The value of δ is initially chosen to reveal all control flow statements depending on d.

A run-time scheduler typically operates with upper bounds of execution times for the units to be scheduled. In our further equations we therefore use only the upper bounds of the execution times, i.e. $\max(t_X(d))$, which we abbreviate by t_X for better readability.

4.2 Dynamic impact analysis

The structure of the application is analyzed and its dynamic behavior is characterized. The main assumption of our precomputation technique is that changes in resource requirements of an application can be observed through changes in values of software variables in that application, which we call **parameters**. For dynamic applications we in general observe all variables $\mathbf{v}$ that read the input data $\mathbf{d}$, the code statements $\xi_{\mathbf{v}}$ these variables $\mathbf{v}$ control, and the changes $\frac{\partial r}{\partial \mathbf{v}}$ in the resource requirements r as the result of changes in $\mathbf{v}$. When the dynamic impact analysis is completed, the variables in $\mathbf{v}$ can be ordered by their impact on the resource requirements:

$$\mathbf{v} = \{v_1, \dots, v_k\}, \text{ where} \tag{2}$$

$$v_1 = f_1(\mathbf{d}), \dots, v_k = f_k(\mathbf{d}), \tag{3}$$

$$r = f_r(v_1, v_2, \dots, v_k), \text{ and} \tag{4}$$

$$\frac{\partial r}{\partial v_1} > \frac{\partial r}{\partial v_2} > \dots > \frac{\partial r}{\partial v_k} \tag{5}$$

Variables in $\mathbf{v}$ represent candidates for precomputation. Once their values are known for a given input $\mathbf{d} = \mathbf{a}$, the resource requirements $r(\mathbf{a})$ can be estimated according to f_r.

For our special case we need to find an ordering of the revealed control flow variables, i.e. parameters $\mathbf{v} = \{e_0, e_1, e_2, e_3, e_4, e_5\}$, according to their impact on the overall execution time of Algorithm 3. We apply dynamic impact analysis proposed by [7]. The method is fully automatic and based on static timing analysis of application code. For each variable v, influence coefficient $IC(v)$ is calculated, which represents the maximum possible variation (in cycles) caused by the different values of the variable v on the WCET of the application. Practically, this is done by traversing the abstract syntax tree (AST) of the program in the backward direction (from the end to the beginning) and computing $IC(v)$ in each statement as a sum of its own contribution and the maximum of $IC(v)$ computed for all its successors in the program. The first statement of the program will yield the ICs computed for each possible variable. Another profiling based method exists, namely the one in [6], which aims at soft real time systems. We assume that the resulting ordering of the parameters is $\mathbf{v} = \{e_0, e_5, e_4, e_1, e_2, e_3\}$.

4.3 Precomputation design

In this step the precomputation is designed based on the information gathered in the previous step. We know which parameters $\mathbf{v}$ control the most dynamic parts $\xi_{\mathbf{v}}$ in the application, how they control them and how the changes in $\mathbf{v}$ influence the resource requirements r of the application. To make this knowledge useful in the run-time environment, the upcoming resource requirements r should be estimated in advance, i.e. before the execution of dynamic parts $\xi_{\mathbf{v}}$. To achieve this, the parameters $v_1, v_2, \dots, v_k$ must be precalculated at the earliest possible position in the application code and remain stable until and throughout the execution of $\xi_{\mathbf{v}}$. This is done by duplicating the statements that update $\mathbf{v}$ and inserting them early in the code as a basis for precomputation. The location for the placement of the duplicate code will be discussed in the next section. From the precomputed values of v and the estimated resource requirements for these values, the resource estimation function f_r is specified that can be used by the run-time resource manager.

ALGORITHM 4: Initial precomputation code

```
Input: $d$, dynamic data
Initialize loop iterators $i$, $j$ and counters $c_A - c_I$
while $e_0(i, d)$ do
    $i = A(i)$;
    $c_A++$;
    if $e_1(i, d)$ then
        $c_B++$;
    end
    else if $e_2(i, d)$ then
        $c_C++$;
        if $e_3(i, d)$ then
            $c_D++$;
        end
        else
            $c_E++$;
        end
    end
    else if $e_4(i, d)$ then
        $c_F++$;
        continue/break;
    end
    else
        $c_G++$;
    end
    while $e_5(j, d)$ do
        $j = H(j)$;
        $c_H++$;
    end
end
```

Algorithm 4 shows the initial precomputation code for the dynamic loop in Algorithm 3. This code is the exact copy of the control flow of Algorithm 3 with added counters c_A,...,c_I, which count the number of times the application visits a particular level of the control flow hierarchy. These counters are used by f_r to estimate r, in this case the required number of CPU cycles. When precomputation has completed and the values of the precomputation counters c_A,...,c_I are known, the total execution time of the code in Algorithm 3 can be estimated as the sum of the products of each counter with the execution times of the associated code block(s) and evaluation expression(s) (Equation 6). For a given counter c_Y, its associated code blocks are all code blocks at the same level of the control flow hierarchy in the original application code, which are executed the same number of times as the counter is incremented. The associated evaluation expressions are all evaluation expressions that are executed to access that level of hierarchy from the level above it. If an evaluation expression must be evaluated to FALSE to access a given level, we mark it by a horizontal bar above it. See the lists below Equation 6. We denote the application execution time estimate based on precomputation by t_{PET}. Recall that t_X is the execution time of the code block X.

$$f_r = t_{PET} = \sum_{Y \in \{A,\ldots,H\}} (c_Y \mid d) \cdot t_{c_Y}, \text{ where } t_{c_Y} = \sum_{\text{all } X \text{ associated with } c_Y} t_X \tag{6}$$

$$\begin{array}{llll} c_A : A, e_0 & c_B : B, e_1 & c_C : C, \overline{e_1}, e_2 & c_D : D, e_3 \\ c_E : E, \overline{e_3} & c_F : F, \overline{e_1}, \overline{e_2}, e_4 & c_G : G, \overline{e_1}, \overline{e_2}, \overline{e_4} & c_H : H, I, e_5 \end{array}$$

To compare, a safe time estimate without precomputation, i.e not exploiting the actual run-time value of input d, must be based on the slowest possible execution of the nested loop for any value of d. Let d_{WCET} denote the value of d that produces such a worst case condition. Then, the worst case execution time t_{WCET} can be estimated by exchanging $(c_Y \mid d)$ in Equation 6 by the worst case count of each code block and evaluation expression $(c_Y \mid d_{WCET})$. Equation 7 below shows the result.

$$t_{WCET} = \sum_{Y \in \{A,\ldots,H\}} (c_Y \mid d_{WCET}) \cdot t_{c_Y}, \text{ where } t_{c_Y} = \sum_{\text{all } X \text{ associated with } c_Y} t_X \tag{7}$$

To evaluate these two estimates for the execution time of the application, we need to compare them with the actual execution time, t_{ET}. As mentioned in Subsection 4.1, t_X in Equations 6 and 7 above represent the upper bounds of the execution times of the code blocks and evaluation expressions. Thus, both estimates are an overestimation of the actual execution time t_{ET}, bounded by the maximum allowed execution time difference δ. In addition, the worst case estimate t_{WCET} is based on the maximum possible execution counts $(c_Y \mid d_{WCET})$, which results in further overestimation in all cases except when the worst case input value $d = d_{WCET}$ occurs. t_{PET} avoids this overestimation at a price of precomputation overhead.

To find the execution time added by precomputation, we compare Algorithms 4 and 3 and observe that the control flow structure of the precomputation code is identical to the control flow structure of the original nested loop, and all code blocks have been replaced by a simple counting operation that takes time 1, except blocks A and H that calculate the loop counters i and j. Thus, Equation 6 can be used as a basis for calculation of the precomputation execution time ($t_{\text{precomputation}}$). We add the counting operation to Equation 6 and update the associated block lists and obtain precomputation execution time as given by Equation 8 below. Note that this is the overhead for the non-optimized code.

$$t_{\text{precomputation}} = \sum_{Y \in \{A,\ldots,H\}} (c_Y \mid d) \cdot t'_{c_Y}, \text{ where } t'_{c_Y} = 1 + \sum_{\text{all } X \text{ associated with } c_Y} t_X \tag{8}$$

$$\begin{array}{llll} c_A : A, e_0 & c_B : e_1 & c_C : e_1, \overline{e_2} & c_D : e_3 \\ c_E : \overline{e_3} & c_F : \overline{e_1}, \overline{e_2}, e_4 & c_G : \overline{e_1}, \overline{e_2}, \overline{e_4} & c_H : H, e_5 \end{array}$$

4.4 Insertion of precomputation

In this step the complete precomputation code is inserted into the application code. The location should be selected before the dynamic parts of the code. There is a requirement that the parameters remain stable between the precomputation code and the dynamic parts they influence. Recall that our precomputation method assumes that the only changes to variables come from the calculations in the tasks. The input data are read only at the beginning of a task graph. Since the method duplicates all statements that update a parameter to an earlier position in the code, even though the parameter changes, its precomputed value will always be correct. However, the number of duplicated statements should be minimized to reduce the overhead of precomputation.

The second consideration that needs to be taken into account when searching for the optimal location of the precomputation code is the response time of the resource management mechanism. The reaction times of run-time resource management mechanisms vary depending on the mechanism used and on the implementation platform. A run-time scheduler in system scenario framework has earlier been reported to have response time of 0.25 ms on a 200 MHz processor [16], p.148. The scheduler monitors the values of parameters and identifies the active scenario. When the scenario has changed, a rescheduling is performed.

4.5 Optimization

This step performs optimization of the precomputation code. Ideally, precomputation code should be merged as much as possible into the application's own code to reduce the overhead of precomputation. In some cases it is possible, while in others it might not be possible. It depends on the code structure and compiler optimizations used/allowed in a given hard real time system. Both cases are allowed as long the precomputation overhead and gain are acceptable. Precomputation code verification is done after optimization.

4.6 Profiling and evaluation

Evaluate the effectiveness of the precomputation on a representative set $\mathbf{D}$ of N input data vectors $\mathbf{d} \in \mathbb{R}_n$. Identify precomputation gain G and precomputation overhead O and also the conditions at which the overhead becomes unacceptable.

We first define precomputation gain and precomputation overhead for a given data input $\mathbf{d}$. Precomputation overhead O represents the resource requirements added to the application by the precomputation code:

$$O = r_{\text{precomputation}} \mid \mathbf{d} \tag{9}$$

Precomputation gain represents the efficiency of precomputation, or the number of times the resource requirements of the application decreases by using precomputation, taking into consideration the overhead of precomputation. We define precomputation gain G as the ratio between the resource requirement estimate based on the worst case input $\mathbf{d}_{WC}$ and resource requirement estimate calculated from precomputation augmented with precomputation overhead:

$$G = \frac{f_r \mid \mathbf{d}_{WC}}{f_r \mid \mathbf{d} + r_{\text{precomputation}} \mid \mathbf{d}} \tag{10}$$

With this definition, precomputation gain equals 1 when the resource requirement estimates with and without precomputation are the same, when taking into consideration the overhead of precomputation. Precomputation gain is more than 1 when precomputation-based resource requirement estimate with added precomputation overhead is less than worst-case-input based resource requirement estimate, so the resources can be saved by using precomputation. Precomputation gain is less than 1 when precomputation-based resource requirement estimate with added precomputation overhead is bigger than worst-case-input based resource requirement estimate, meaning that precomputation is counter-productive.

To evaluate the gain and the overhead of precomputation for an embedded application that is continuously rerun with different input data, we profile the application with inserted precomputation with the representative data set $\mathbf{D}$ and compute the average precomputation overhead $\bar{O}$ and the average precomputation gain $\bar{G}$ by Equations 11 and 12 respectively:

$$\bar{O} = \frac{1}{N} \sum_{\mathbf{d} \in \mathbf{D}} r_{\text{precomputation}} \mid \mathbf{d} \tag{11}$$

$$\bar{G} = \frac{1}{N} \sum_{\mathbf{d} \in \mathbf{D}} \frac{f_r \mid \mathbf{d}_{WC}}{f_r \mid \mathbf{d} + r_{\text{precomputation}} \mid \mathbf{d}} \tag{12}$$

For our case application, precomputation gain and precomputation overhead are calculated in terms of execution time. Precomputation overhead O corresponds to the precomputation execution time $t_{\text{precomputation}}$ in Equation 8. The average precomputation gain can then be computed by Equation 13 by profiling the application with N data vectors $\mathbf{d} \in \mathbb{R}_n$ from a representative set $\mathbf{D}$:

$$\bar{O} = \frac{1}{N} \sum_{d \in D} t_{\text{precomputation}} \tag{13}$$

Precomputation gain is the ratio of execution time estimate for the worst case input t_{WCET} (Equation 7) and the sum of execution time estimate based on precomputation t_{PET} (Equation 6) and precomputation execution time $t_{\text{precomputation}}$. The average precomputation gain for the representative data set $\mathbf{D}$ is given by Equation 14:

$$\bar{G} = \frac{1}{N} \sum_{d \in D} \frac{t_{WCET}}{t_{PET} + t_{\text{precomputation}}} \tag{14}$$

Note that the accuracy of the estimated precomputation gain $\bar{G}$ and precomputation overhead $\bar{O}$ depend on the accuracy of the applied data model.

4.7 Check if precomputation overhead $\bar{O}$ is acceptable

In this step precomputation overhead $\bar{O}$ is evaluated. This can be done by specifying a percentage γ that the total resource requirements of the application can increase due to precomputation. Specification of γ requires a global system evaluation with given overall resource constraints. If $\bar{O}$ satisfies this requirement, precomputation design is completed and the next step is to evaluate its effectiveness. If not, an iterative approach as indicated in Figure 1 can be applied to simplify the precomputation code.

4.8 Remove precomputation of parameters with least dynamic impact and most overhead

From the dynamic impact analysis in Subsection 4.2 we have a list of parameters ordered by their dynamic impact on the resource requirements, see Equation 2. To lower the overhead of precomputation, we iteratively remove precomputation of parameters having the least dynamic impact on the resource requirements. If this does not help, we try removing parameters that have the largest impact on the precomputation overhead. Each time the precomputation is simplified, the overhead is reduced, but so is also the gain. The reason for this is that the resource requirement estimates of the removed parts must be reset to the worst case to avoid underestimation. The steps in Subsections 4.4 - 4.7 are repeated until the precomtputation design is accepted or the precomputation is found inefficient, in which case it is removed.

In our case we adjust the precomputation loop structure by hiding some of the control flow to obtain acceptable gains and overheads. Using the list of parameters $\mathbf{v} = \{e_0, e_5, e_4, e_1, e_2, e_3\}$ from Subsection 4.2, ordered by decreasing dynamic impact on the execution time of the application, we decide to hide the nested if structure (branches e_1, e_2 and e_3) keeping only the branch e_4 that steers the continue/break statement. The resulting precomputation code is shown in Algorithm 5. The execution time estimate based on the simplified precomputation is given by Equation 15, and the overhead of the precomputation in Algorithm 5 is presented in Equation 16. Note that the t_{PET} estimate for the hidden code structure defaults to the worst case values.

ALGORITHM 5: Final precomputation code

```
Input: $d$, dynamic data
Initialize loop iterators $i, j$ and counters $c_A, c_F, c_H$
while $e_0(i, d)$ do
    $i = A(i)$;
    $c_A + +$;
    if $\overline{e_1}$ & $\overline{e_2}$ & $e_4(i, d)$ then
        $c_F + +$;
        continue/break;
    end
    while $e_5(j, d)$ do
        $j = H(i)$;
        $c_H + +$;
    end
end
```

$$t_{PET} = \sum_{Y \in \{A,F,H\}} (c_Y \mid d) \cdot t_{c_Y} + (c_A - c_F) \cdot t_{\text{hidden}}, \text{ where}$$
$$t_{\text{hidden}} = \max\left(t_{c_B}, t_{c_C} + t_{c_D}, t_{c_C} + t_{c_E}, t_{c_G}\right) \text{ and } t_{c_Y} \text{ is given by Eq. 6} \tag{15}$$

$$O = \frac{1}{N} \frac{\sum_{d \in D} \left(\sum_{Y \in \{A,F,H\}} (c_Y \mid d) \cdot t'_{c_Y}\right)}{\sum_{d \in D} t_{WCET}}, \text{ where } t'_{c_Y} \text{ is given by Eq. 13} \tag{16}$$

4.9 Check if precomputation gain $\bar{G}$ is acceptable

In profiling and evaluation step in Subsection 4.6, precomputation gain $\bar{G}$ was calculated, and now a decision must be made if this gain is acceptable. Recall that precomputation gain is defined as the average number of times resource requirements of the design decreases when precomputation is used. We specify a threshold l representing the minimal average number of times the resource requirements of the application must decrease when applying precomputation. The number l should be selected based on a global system evaluation with a given run-time resource manager. If precomputation gain $\bar{G}$ is above the threshold l, the precomputation is effective and improves the resource utilization of the system. The precomputation design is accepted. On the other hand, if precomputation gain $\bar{G}$ is below the threshold l, the precomputation is inefficient in the application at hand and is not recommended. This happens in cases when precomputation of parameters requires itself too much resources and cannot be optimized.

4.10 Implementation remarks

Note that all steps in our algorithm is safe in the sense that they ensure that the resource requirements are never underestimated. For each code block and evaluation statement an upper bound of the resource requirements is used, while the precomputed counter values $(c_X \mid d)$ are always equal to the actual number of activations of each code block and evaluation expression X. Also each time we remove a parameter from precomputation, we set the resource requirement estimate to the worst case for all the blocks and statements in the corresponding structure in the original code. Thus this method respects the resource limits of the system and can handle arbitrary worst-case data situations. However it will only be efficient when worst-case situations appear seldom. To compare, classic static methods consider all data situations as worst-case and they run most efficiently in the worst case. Precomputation makes use of the dynamism in the data and runs more efficiently than classical methods in all other data situations where the required resources are less than the worst case (minus the overhead of the precomputation code).

However, a good model of the expected data is necessary at design time to obtain realistic estimates of precomputation gain G and precomputation overhead O to evaluate precomputation efficiency. This works well for stationary data with constant statistical properties. Application of precomputation in case of non-stationary data should be investigated by further research.

5 EXPERIMENTAL RESULTS

In this section we present experimental results from implementation of run-time precomputation of data-dependent parameters in a real-life biomedical application, STLEC, on the CoolBio biomedical signal processing (BSP) platform [12]. STLEC performs a continuous estimation of the largest short-term Lyapunov exponent (STLmax) from human EEG. STLEC is used in an epileptic seizure predictor [13, 15] and other related systems [14]. 32 EEG sensors perform brain wave measurement with a 200 Hz sample rate. For each sensor, 2048 samples (10.24 seconds) are collected into a sample set on which a STLmax calculation is performed. Generally, Lyapunov exponents are used in many practical applications dealing with phenomena that are modeled as physical dynamical systems, when the system equations are not explicitly known. Lyapunov exponents are defined as the average exponential rates of divergence or convergence of nearby orbits in phase space representations of such dynamical systems.

STLEC is characterized by a varying execution time depending on the input EEG, see histogram in Figure 2. Profiling on a CoolBio BSP with a 32-channel 6 hrs long EEG recording from Arizona State University (ASU)[1] have shown that the application requires $55 \cdot 10^6$ clock cycles in the best case and $113 \cdot 10^6$ clock cycles in the worst case.

Recall that the proposed methodology is aimed at dynamic applications, i.e. where the change in resource requirements to be optimized is caused by change in the input values of the application. It is also assumed that the change in the resource requirements can be observed through values of internal software variables that we call parameters. Precomputation of these parameters will be efficient if the typical input values of the application do not produce the (near) worst case conditions in the dynamic part of the code.

STLEC utilizes a variant of the well-known fixed evolution time program of [21] and has a dynamic code built up from control structures presented in Algorithm 3. To find the largest Lyapunov exponent, the application takes in a segment of EEG (2048 samples) and from it constructs a phase space model of the brain system. It then selects a reference orbit in the model and spends most of its time on searching for the nearest neighbor orbit. At each time step, the point on a nearest neighbor orbit must satisfy two conditions: 1) Minimum distance to the reference orbit above the noise level, and 2) Minimum change of orientation of the neighbor orbit. The search is done iteratively, where only points inside a cone with a selected radius and apex angle are searched. If the desired point is not found, the radius is increased. If still not found, the angle is increased while the radius is reset to the original value. This procedure is repeated several times. If no point satisfying 1) and 2) is found, the point from the previous time step is extrapolated in the direction of the orbit found so far.

[1]Provided by Prof. Leonidas Iasemidis that was working at ASU at that time. The origin of the data is confidential due to an NDA with ASU.

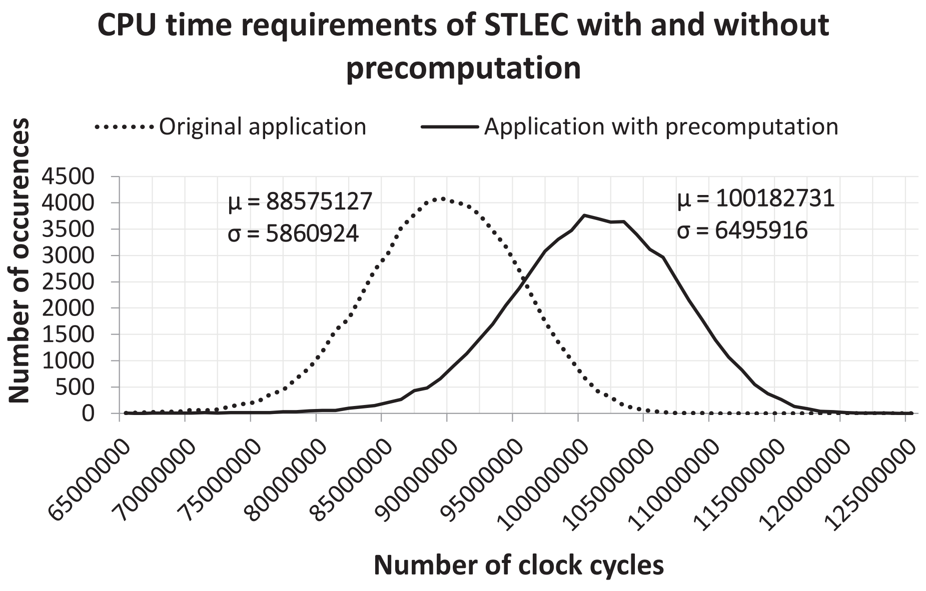

Fig. 2. Histogram over clock cycles for 60,000 runs of STLEC application on CoolBio BSP platform with and without precomputation

We identified three parameters that have the greatest impact on the execution time of STLEC, sorted in the descending order of impact:

- The total number of points in the phase space representation that can be searched to find the nearest orbit. This parameter depends on the number of data points in each EEG segment and the dimension of the phase space representation, which are typically fixed, so it does not have real dynamic impact.
- The number of iterations of the search procedure at each time step of the algorithm before the point that satisfies 1) and 2) is found.
- The number of points in each iteration of the search with distances above the noise level and below the selected radius. They satisfy the condition 1) above and undergo an additional angle check 2) that requires extra time.

The last two parameters depend on the input EEG that is not known in advance. They are the candidates for our precomputation methodology. We attempted to implement precomputation of both parameters using the proposed method, Subsections 4.1-4.8. However the second parameter in the list above was finally removed as incurring too much overhead, although it had greater dynamic impact than the third parameter with the original settings of the algorithm. To precompute this parameter at a given time step of the algorithm, we would need to perform a complete search of the next step, meaning that we would double the amount of clock cycles and memory requirements as compared with the case with no precomputation.

In the final experiment we reduced the impact of the second parameter by modifying the algorithm settings. We reduced the number of searched apex angles from three to a single angle. The original STLEC had also a reduced number of angles as compared to [21]. Note that this may affect the quality of the STLmax calculation.

The precomputation code of the third parameter, denoted *preNrAngleChecks*, is shown in Algorithm 6 with original comments from [21].

ALGORITHM 6: STLEC precomputation code

```
//LOOK FOR REPLACEMENT POINT
//ZMULT IS MULTIPLIER OF SCALMX WHEN GO TO LONGER DISTANCES
...
$preNrAngleChecks = 0$;
if $zmult < mult$ then
    $prescalmx = dnewmx * bb*(double)(zmult + 1)$;
end
else
    $prescalmx = dnewmx * bb$;
end
//SEARCH OVER ALL POINTS
...
//LOOK FURTHER AWAY THAN NOISE SCALE, CLOSER THAN ZMULT*SCALMX
$dnew\_check = dnew < scalmn$;
if $dnew\_check || dnew > prescalmx$ then
    $preNrAngleChecks + +$;
end
$dnew\_check = dnew\_check || dnew > scalmx$;
```

We evaluated the design by cycle accurate simulation in Target Compiler Technologies' Checkers tool [2]. CoolBio BSP has 4 power modes [12]: 1) Data collection mode (DC): only the periphery domain and the required number of data memory banks are on, running at 0.4 V and 0.7 V respectively; the program memory is in retention mode and the rest of the system is off; 2) Low power mode (LP): the processor and periphery are on, running at 0.4 V, part of the program and data memory are on, running at 0.7 V and the unused memory banks are off; 3) High performance (HP) mode: All components are running at 1.2 V, and the unused memory banks are off; 4) Sleep mode: all power domains are off and no internal clock is running. The authors provide only numbers for dynamic power consumption for the three active modes. They comment that the processor is not optimized for deep-sleep mode, but for a power efficient data collection mode. We assume a static power consumption of 5% of the dynamic power consumption, which is reasonable for standard 90 nm LP process and use the numbers in the paper for the normalized leakage scaling in low power mode and data collection mode. Table 1 shows the result of this calculation. Unfortunately, it seems that low power mode is not energy-efficient for the case with full functionality due to its low frequency. It is probably intended for power-gated operation. Since the primary goal of this experiment is to consider DVFS gains and overheads, we do not include power gating/reduced functionality operation. Hence, we exclude low power mode from consideration and add instead two interpolated modes between the low power mode and the high performance mode. See Table 1 for characterization of the interpolated modes.

The application starts in one of the considered DVFS modes when a full sample set of EEG values to perform STLmax estimation has arrived, runs until the estimation is done, switches to the data-collection mode and stays in this mode until the deadline, when a new set of EEG values arrives and the process is repeated. We compare the following three cases:

(1) Static DVFS scheduling (STA). The power mode is selected before the application runs based on worst case clock cycles of the application and the time to deadline.
(2) Dynamic DVFS scheduling (DYN). The power mode is selected at the activation of each task at run-time based on worst case clock cycles of the task and its allocated time. The time is

[2]Target Compiler Technologies was acquired by Synopsys in February 2014

Table 1. Characterization of CoolBio[12] original power modes and interpolated power modes

Power modes	DC Mode	LP Mode	Mode 1	Mode 2	HP mode
Frequency [MHz]	1	1	38	69	100
Voltage [V]	0.4/0.7	0.4/0.7	0.7	0.95	1.2
Dynamic power [μW]	13	1792	1881	6276	14500
Estimated static power [μW]	0	56	145	435	725
Total power [μW]	13	1848	2026	6711	15225
Total energy per cycle [pJ/cycle]	13	1848	53	10	152

Table 2. Comparison of energy consumption per STLEC run between STA - static scheduler, DYN - simple dynamic scheduler, and DYNPRE - same as DYN, but with precomputation of data-dependent parameters

STLEC	STA			DYN			DYNPRE		
Energy/run [mJ]	Min.	Typ.	Max.	Min.	Typ.	Max.	Min.	Typ.	Max.
Processing energy	8.38	13.54	17.23	2.95	7.49	11.14	3.32	5.60	8.75
DVFS switching [μJ]		1.17		0.5	3.00	5.16	0.4	0.27	2.09
Dynamic scheduling		0		1.36	2.27	2.76	1.36	2.27	2.76
Total energy	8.38	13.54	17.23	4.31	9.75	13.87	4.68	7.87	11.48

allocated to each task before the application runs, based on the worst case clock cycles of the task. When the task is completed, time slack to its deadline is used in the power mode selection of the next task.

(3) Dynamic DVFS scheduling with precomputation (DYNPRE). Same scheduler as 2 is used, but for the dynamic task with precomputation the precomputed clock cycles are used instead of worst case clock cycles to select the power mode. The overhead of precomputation is included in the precomputed cycles

The deadline for a single channel equals the time it takes to acquire the full set of EEG samples, 10.24 s. There are 32 EEG channels, while the schedulers in the list can process 3 channels within the deadline. We set deadline for a single channel equal to 10.24 s/3 = 3.41 s.

We evaluated average energy consumption of STLEC application for the three scheduling cases. Totally, 60,000 runs of STLEC were done and the number of tasks varied between min/typ/max = 356/596/724. Table 2 presents the results of this evaluation. In the table, processing energy is the energy that is consumed by the application's tasks when run with the DVFS settings selected by the applied scheduler. In the DYNPRE case, processing energy also includes the overhead of processing the precomputation code of Algorithm 6.

The remaining overheads are rough estimates. DVFS switching overhead is the extra energy that is consumed due to switching of power modes. We use same approach and switching times as the authors in [22] used for the SAM4L processor. The overhead is calculated as the product of switching time and the power consumption of the previous power mode. Switching times for voltage upscaling and voltage downscaling are 16.4 μs and 1.25 μs, respectively. Note that precomputation reduces DVFS switching overhead. This is caused by reduction of the number of switches. Another observation that we made is that precomputation changes the distribution of DVFS switching energy. While in case of DYN, switching energy has a distribution close to normal, with average of 3.00 μJ and standard deviation of 0.68 μ, the switching energy in DYNPRE case has a completely

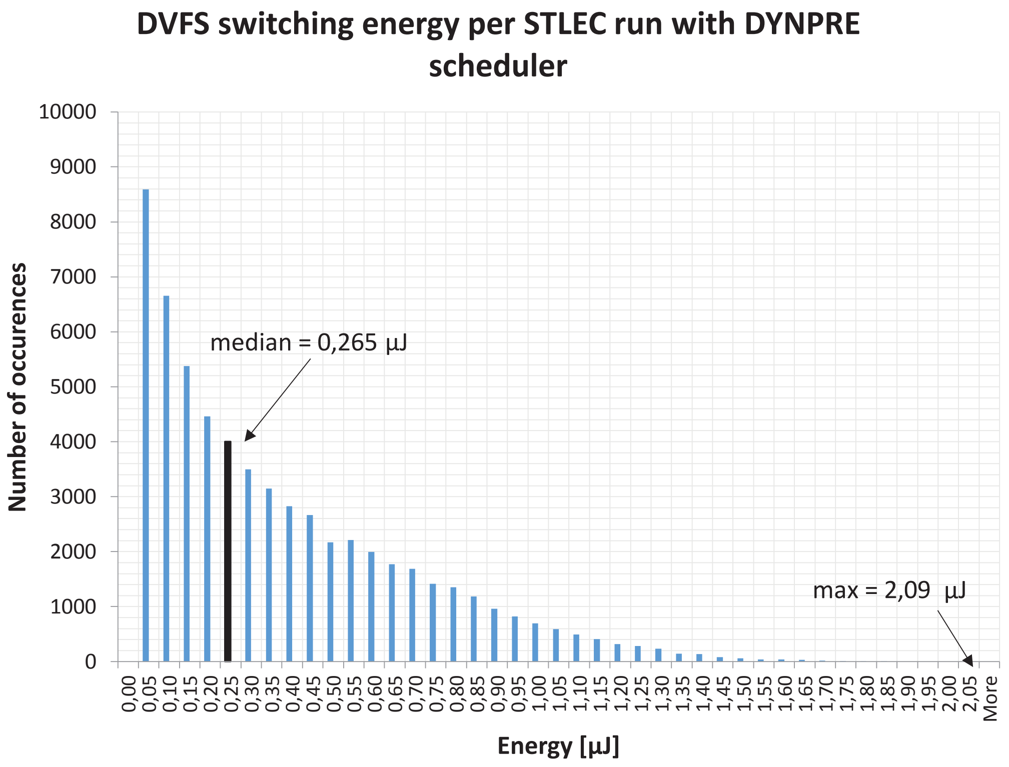

Fig. 3. DVFS switching energy per run of STLEC application with DYNPRE scheduler

different form, see Figure 3. We use median values for the typical switching energy for DYNPRE case in Table 2.

Dynamic scheduling overhead considers the extra time it takes to reschedule the tasks at run-time. We used the response time of a run-time scheduler in [16], 250 μs, to calculate this overhead. We assumed that the scheduler is activated once per task.

The overhead of the precomputation is presented in Figure 2. Precomputation increases the cycle count of STLEC application by 13.1% on average.

Figure 4 shows the distributions of the total energy consumption per run of the application, including all the investigated overheads. The figure shows that the run-time precomputation of a single data-dependent parameter used by a dynamic DVFS scheduler reduces the energy consumption of the system by 40% on average compared to static DVFS scheduler, and by 18% on average compared to dynamic DVFS scheduler without precomputation.

6 DISCUSSION

The results in Section 5 show that precomputation has a positive effect in a real system, taking into consideration the overheads of processing extra precomputation code, rescheduling the tasks and up- or downscaling the supply voltage.

In this application the final precomputation is done for a single parameter and gives good results. Generally, it is desirable to keep the number of parameters that are precomputed low in order to reduce the overall precomputation overhead. It can therefore be expected that some of the parameters will be pruned away from precomputation code after the profiling and evaluation step in the methodology.

Although we only considered a single application in our experiments, the results are nevertheless applicable to other hard real-time dynamic applications where execution time varies due

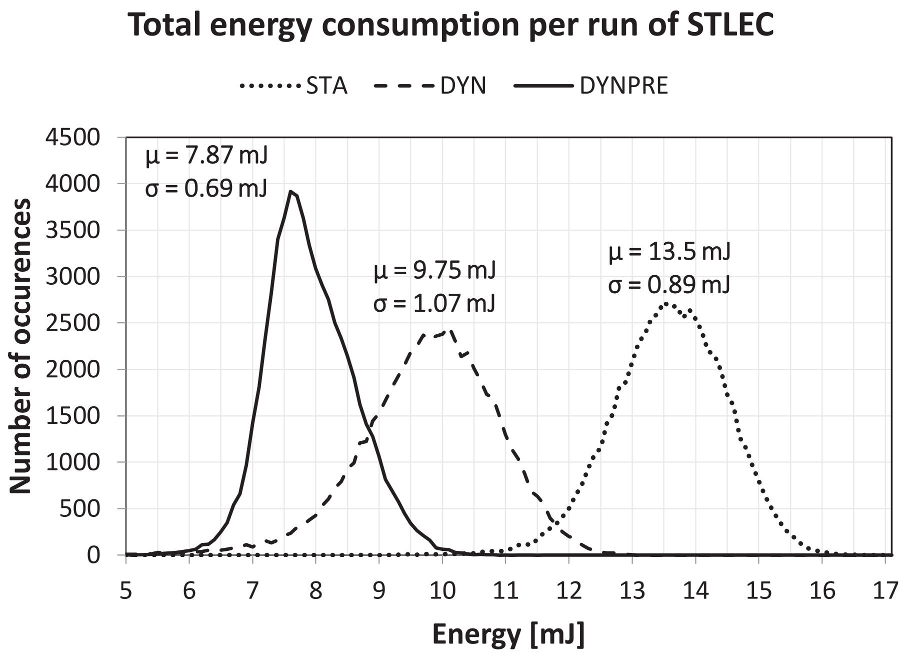

Fig. 4. Distribution of total energy per run of STLEC application with STA, DYN and DYNPRE schedulers

to data-dependent control structures such as loops and branches. The exact percentage of saved energy will be different, as it depends on the actual control structure, target platform and input data, but precomputation has a good potential to reduce the system energy consumption when the application rarely runs in the worst case conditions.

Automatic precomputation of data-dependent parameters should be investigated in further research. Since the current method is based on replication of the dynamic control structures and addition of simple counters, the automation seems feasible. Integration with compiler and co-operation with other compiler optimization techniques needs to be investigated.

7 CONCLUSIONS

We have presented a method for run-time precomputation of data-dependent software variables, called parameters, having the most impact on system resource requirements. The method is aimed at dynamic embedded hard real-time systems containing run-time resource management mechanisms that can monitor parameters and exploit the knowledge of their resource impact for better resource utilization. The efficiency of the proposed method depends on the application structure and the statistical properties of the input data, particularly, how often the system requires the maximum resource usage. Experiments with a real life algorithm and real data show a significant energy gain for data-dependent parameter precomputation.

ACKNOWLEDGMENTS

The authors wish to thank Prof. Leonidas Iasemidis at Louisiana Tech University and Prof. Konstantinos Tsakalis at Arizona State University for possibility to experiment with their STLEC code and EEG data and their valuable feedback and discussions. We also thank Dr. Shivkumar Sabesan, Dr. Balu Krishnan and Aaron Faith for supporting the experiments.

REFERENCES

[1] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 7 (Jan. 2016), 34 pages. https://doi.org/10.1145/2808231

[2] T-F. Chen and J-L. Baer. 1995. Effective hardware-based data prefetching for high performance processors. *IEEE-TransComp* 44, 5 (May 1995), 609–623. https://doi.org/10.1109/12.381947

[3] J. Engblom. 1999. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*. 46–55. https://doi.org/10.1109/RTTAS.1999.777660

[4] Iason Filippopoulos, Francky Catthoor, and Per Gunnar Kjeldsberg. 2013. Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios. *Transactions on Biomedical Circuits and Systems* 17, 3-4 (September 2013), 669–692. https://doi.org/10.1007/s10617-014-9145-6

[5] John .W.C. Fu, Janak .H. Patel, and Bob .L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 24th annual international symposium on Microarchitecture (MICRO 25)*. IEEE, Los Alamitos, CA, USA, 102–110.

[6] S.V. Gheorghita, T. Basten, and H. Corporaal. 2006. Profiling Driven Scenario Detection and Prediction for Multimedia Applications. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS)*. IEEE, Los Alamitos, CA, USA, 63–70.

[7] S.V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. 2005. Automatic Scenario detection for improved WCET estimation. In *Proceedings of the 42nd Design Automation Conference*. ACM Press, NY, USA, 101–104.

[8] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. 2009. System-scenario-based design of dynamic embedded systems. *Transactions on Design Automation of Electronic Systems (TODAES)* 14, 3 (January 2009), 1–45. https://doi.org/10.1145/1455229.1455232

[9] M. Grannaes, M. Jahre, and L. Natvig. 2010. Multi-level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matchins. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEACâĂŹ10)*. Springer-Verlag, Berlin, Heidelberg, 247–261.

[10] E. Hammari, F. Catthoor, P.G. Kjeldsberg, J. Huisken, K. Tsakalis, and Iasemidis L. 2012. Identifying Data-Dependent System Scenarios in a Dynamic Embedded System. In *Proceedings of the 2012 International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA'12)*. CSREA Press, Las Vegas, NV, USA, 70–77.

[11] J.L. Hennesy and D.A. Patterson. 2012. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, Burlington, Massachusetts, USA.

[12] Jos Hulzink, Mario Konijnenburg, Maryam Ashouei, Arjan Breeschoten, Torfinn Berset, Jos Huisken, Jan Stuyt, Harmke de Groot, Francisco Barat, Johan David, and Johan Van Ginderdeuren. 2011. An ultra low energy biomedical signal processing system operating at near-threshold. *Transactions on Biomedical Circuits and Systems* 5, 6 (December 2011), 546–554. https://doi.org/10.1109/TBCAS.2011.2176726

[13] L.D. Iasemidis, Deng-Shan Shiau, Panos M. Pardalos, Wanpracha Chaovalitwongse, K. Narayanana, Awadhesh Prasad, Konstantinos Tsakalis, Paul R. Carney, and J. Chris Sackellares. 2005. Long-term prospective on-line real-time seizure prediction. *Clinical Neurophysiology* 116, 3 (March 2005), 532–544.

[14] L. D. Iasemidis. 2011. Long-term prospective on-line real-time seizure prediction. *Neurosurg Clin N Am* 22, 4 (October 2011), 489–506.

[15] Leon D. Iasemidis, Deng-Shan Shiau, Wanpracha Chaovalitwongse, J. Chris Sackellares, Panos M. Pardalos, Jose C. Principe, Paul R. Carney, Awadhesh Prasad, Balaji Veeramani, and Konstantinos Tsakalis. 2003. Adaptive epileptic seizure prediction system. *Transactions on Biomedical Engineering* 50, 5 (May 2003), 616–627.

[16] Z. Ma, P. Marchal, D.P. Scarpazza, P. Yang, C. Wong, J.I. Gomez, S. Himpe, C. Ykman-Couvreur, and F. Catthoor. 2007. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms*. Sringer, P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

[17] R. Nair. 1995. Optimal 2-bit branch predictors. *IEEETransComp* 44, 5 (May 1995), 698–702. https://doi.org/10.1109/12.381956

[18] Kyle J. Nesbit and James E. Smith. 2004. Data Cache Prefetching Using a Global History Buffers. In *10th International Symposium on High Performance Computer Architecture (HPCA âĂŹ04)*. IEEE, Madrid, Spain, 96–106.

[19] Mladen Skelin, Marc Geilen, Francky Catthoor, and Sverre Hendseth. 2014. Worst-case Throughput Analysis for Parametric Rate and Parametric Actor Execution Time Scenario-Aware Dataflow Graphs. In *Proceedings of the 1st International Workshop on Synthesis of Continuous Parameters (SynCoP)*. Grenoble, France, 65–79.

[20] Alan Jay Smith. 1982. Cache Memories. *Comput. Surveys* 14, 3 (September 1982), 473–530. https://doi.org/10.1145/356887.356892

[21] A. Wolf, J.B. Swift, H.L. Swinney, and A. Vastano. 1985. Determining lyapunov exponents from a time series. *Physica 16D* 16 (1985), 285–317.

[22] Yassin Y., Kjeldsberg P.G., Perkis A., and Catthoor F. 2016. Dynamic hardware management of the H264/AVC encoder control structure using a framework for system scenarios. IEEE, Limassol, Cyprus, 222–230.

[23] T-Y. Yeh and Y. N. Patt. 1991. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture (MICRO 24)*. ACM, New York, NY, USA, 51–61.

[24] N. Zompakis, I. Filippopoulos, P.G. Kjeldsberg, F. Catthoor, and D. Soudris. 2014. Systematic Exploration of Power-Aware Scenarios for IEEE 802.11ac WLAN Systems. In *Proceedings of the 17th EUROMICRO Conference on Digital System Design (DSD)*. IEEE, Verona, Italy, 28–35.