Abstract — The past few years, there has been an immense development in the field of embedded systems. These systems can perform significant number of computations and execute complex applications, which are either specific or general purpose oriented. Examples of these systems are handheld devices etc. The only common factor all these devices have, is the power needed to supply the circuitry to work. At this point there are many methods and techniques suggested by academic and industrial community, to address this problem, each one at a different level of the architecture of the system. In this paper, we emphasize on the Power Supply Level, where Dynamic Voltage and Frequency Scheduling Techniques can be implemented, giving related work on this level and also analyzing some of these methods in a more detailed manner.

Index Terms — embedded system, inter / intra dynamic voltage and frequency scheduling techniques, power (energy) dissipation

I. INTRODUCTION

Before we begin exploring the problem of power dissipation in embedded systems, let’s spend some time elaborating on what is thought to be an embedded system nowadays. As it is very well defined in [1], an embedded system is a system that is designed from scratch, to serve a specific purpose. This characteristic predefines the variation of jobs it can handle, in terms of numbers and complexity. In other words it is both an advantage and a disadvantage. This is because as a special-purpose system, it can be configured to present optimal results, when executing the explicit jobs designed to, but this is also its limitation, because it will perform poorly in other, out of its field, jobs. As a result, this kind of systems, are usually small in size (i.e. light-weight) and thus portable. This packet of design also includes limited resources, referring to power, computational capabilities, memory etc.

For example, embedded systems can be considered the everyday mobile phones, the PDAs, medical monitoring devices, sensor systems, videogames, even the laptops. In this paper, we will focus our research on how power dissipation can be addressed in such a system outlined above.

As described in [2], power dissipation in embedded computing systems, is an existing problem, which is gradually becoming the decisive factor in the design and implementation of such systems. The reason is that high power consumption, limits its mobility lifetime, since practically every embedded system, with minor exceptions (e.g. solar, kinetic and wind energy powered systems), need batteries to work.

In addition to the problem of limited power resources, we have the constant anticipation for utmost performance by a small scale computing system such as an embedded system, as it is pointed out in [3]. Here, an oxymoron schema is formed: on the one hand, demand for high performance computation, which consequently rephrases into high power input, which is large power supply units, and on the other hand small packaging-size and weight requirements, which eventually mean lead to minimizing batteries!

A compromising state is being reached so far, that allows light-weight devices to handle a great deal of work, with a considerable performance to size ratio. But requests for better performance at smaller-sized devices, push science to overcome the barrier set by this ratio and place it at a higher level, to meet the expectations of tomorrow’s world. In this momentum, a lot of research has been done, and novel ideas have been presented, that improve this ratio, by various ways. These techniques are to be presented in the next section, for a wider understanding of the different aspects of this problem, and how it can be tackled by a computer architect.

II. POWER MANAGEMENT TECHNIQUES

A. Overview

In this section we will describe various methods presented and tested by the scientific community, with main scope the increase of the performance to power consumption ratio, either by reduction of power consumption with fixed computational performance, increase of performance with fixed power consumption, or by having both factors enhancing the ratio at the same time.

These various techniques, address the problem in the different levels of the architecture of an (light-weight) embedded system. These levels are: (1) Design Level – Number System Level, (2) Power Supply Level (Dynamic Voltage and Frequency Scheduling), (3) CMOS - Gate Level, (4) Register and Instruction Level, (5) Operating System
Level, (6) Memory Level, (7) Peripherals Level, (8) Software Level, (9) Applications Level. We present an overview of these levels and corresponding techniques in the next few paragraphs of this section.

1) Design Level – Number System Level: Starting from the very bottom of the design abstraction of an embedded system, we find the number system. Every digital system, implements a way numbers are represented, so that communication is achieved between the various parts of the system, using the digital signals produced, based on the number system. So it is clear that the correct choice of the number system, affects the various levels of the architecture: the number of operations, the very strength of them and the activity of the data. As presented in [4], this choice can reduce the number of instructions needed to complete tasks. As a consequence, power dissipation can be reduced. So a thorough investigation on the set of applications the system will handle should be made in advance, to help the architects use the most suitable number system for the design of the embedded system.

2) Power Supply Level (Dynamic Voltage Scheduling – DVS, Dynamic Frequency Current Mode Logic - DFS): In this level we have already a given design, i.e. an implementation of the embedded system, and we are varying voltage and frequency - clock of the processor of the system, to reach the desired level of performance, by only using the minimum amount of power needed. Various algorithms have been proposed that realize this methodology and help achieve this goal. This level will be the focus of our paper. So we will not mention any relevant work at this point.

3) CMOS - Gate Level: The CMOS Level of design offers a lot of ground for experimentation to achieve higher performance in lower power levels. An approach – methodology is developed in [5], which provides techniques for either separately optimizing or balancing between throughput and energy consumption, in CMOS Level. One other application, in MOS Current Mode Logic (MCML) is analyzed in [6], for low power and noise signal applications, which demand high security, such as embedded cryptographic processors.

4) Register and Instruction Level: If we move a step above, in the architecture of an embedded system, we reach the Registry Level. An analysis framework is provided in [7], that helps to accurately and efficiently estimate the power dissipation in an embedded system, at the Registry Transfer Level (RTL).

Moving to the Instruction Level, in [8], the authors present an improved decompression engine. In this technique, instructions that have been offline - compressed, decompress on demand, improving the power consumption of the system.

5) Operating System Level: The operating system level is the first of the software levels of the architecture of a computing system. Thus it is responsible for connecting the hardware to the software, i.e. implementing the interface between the jobs to be executed and the actual infrastructure that will carry out these jobs. In this perspective, [9], offers an analysis of real-time operating system (RTOS) power consumption. A different approach is given by [10], where a dynamic thermal management is investigated, and an on-the-fly method of energy consumption estimation is presented, which depends on the applications, services and users. In the same spirit, [11] elaborates on the selection of jobs by the operating system for execution, in order to achieve maximization of the number of execution deadlines and of the total utilization of jobs that finish on deadline, during the operation.

6) Memory Level: Memory in a computing system is both hardware and software component. That is because you can consider the design of memory hierarchy and proceed to changes in the hardware level, i.e. types of memory modules used, interconnections, buses etc, or you can proceed to changes in organization level, where memory structure is given, but you can alter the way data is being saved, retrieved etc (Virtual memory etc). In [12], a new system-level approach is proposed, to model dynamic memory managers and incorporating a power profiling method. Also in [13], a new tool is developed, for estimating power of caches, built inside a united structure – framework, for the design of embedded systems.

7) Peripherals Level: Peripherals can be considered all the devices that surround the processor and main modules of the architecture. Such devices are audio and video devices. In [14] a methodology for estimation of energy consumption in such peripherals is being presented.

8) Software Level: In this level, the various tasks that are to be executed need to be organized in terms of either order, or even code structure, in that way, to produce a more efficient and power-aware code, usually through optimized algorithms. In this philosophy, [15] investigates software scheduling policies, to reduce leakage power in real-time systems. Another novel idea is presented in [16], where clusters of operations and instructions are being mapped to the involved resources to be used, and thus reducing power dissipation. Also in [17], the authors provide a trace-based technique to estimate software power (CPU, memory, busses) and optimization techniques for utilization of the system (software power, code size, performance).

9) Application Level: Complete applications can be optimized on an embedded system, so that power consumption
minimizes for a specific applications set. Reference [18], provides a nice example of application of the Conventional Matched-Field Processing (CMFP) in embedded systems and results of the application to the performance of the system. In [19] an applied design of a portable fingerprint biometric authenticator is presented, with detailed analysis on tradeoffs between security, performance and power issues.

B. Paper Focus

The above examination shows that power dissipation in an embedded system, can be addressed in the different levels of the system’s architecture, with very impressive results in each case. The purpose of this paper is to focus on one of the levels described, so that we investigate further the methods and techniques proposed and provide a more comprehensive study for the particular level.

From the literature survey conducted, we feel that a lot can be shown in the Power Supply Level, where Dynamic Voltage Scheduling and Dynamic Frequency Scheduling, can be used to improve the power dissipation of the system. So our paper will address the problem of power consumption at this one level, providing more detailed analysis of the existing techniques presented by academia and industry, and thus contributing to the better understanding of the specific problem, from the Dynamic Voltage and Frequency Control point of view. The remainder of this paper is organized as follows: In section III, we overview the related work done so far on Dynamic Voltage Scheduling and Dynamic Frequency Scheduling, the past few years. Since this topic is very popular among the ways of reducing the energy consumption in an embedded system, literature is rich of methods and techniques which address the matter at this level. In section IV, we first present the basics of the DVS - DFS techniques, and then we do reviews of six papers, published in world conference proceedings and journals. The review of each paper is based on the material presented from the authors (equations, algorithms, data-results from experiments), but including comments, remarks and conclusions of the writer of this paper. Finally, in section V, general conclusions and suggestions are given for future work.

III. RELATED WORK ON DVS AND DFS

In the Power Supply Level (PSL), as mentioned above, we are varying the supply which the system is working with, i.e. voltage and clock of the processor of the system, to reach the desired level of performance, by only using the minimum amount of power needed. Due to the nature-characteristics of an embedded system, PSL is a decisive factor, for the power consumption of such a system. That is why most of the research presented so far, tackles the problem of power dissipation, through this point of view. This means a lot of theoretical analysis and design implementations can be cited. Most of them, demonstrate methods of Dynamic Voltage Scheduling alone, and others a combination of DVS and Dynamic Frequency Scheduling (DFS). In this section we will present an effort of grouping these various methods, into categories, depending on how they address the matter.

In [20], the algorithm presented, seeks to find voltages that fit each process’ demands, using two key points: no beforehand restriction of the voltage level applied for each process and no fixed limit in the power consumption between processes. An analytical investigation on how many levels to implement in a multiple-voltage DVS system is presented in [21], were analytical solutions for dual-voltage system are derived, as well as efficient numerical methods, for the general case. Also in [22], a novel design is proposed, DVS-FS, which combines DVS and feedback scheduling. The idea is to save CPU power, but also guarantee control performance, which depends on the arrangement of the control task set. In this tone, [23], presents a method using a DVS scheduler and a feedback controller, implementing a DVS-EDF scheduling algorithm. Furthermore, in [24] and [25], the authors address the matter by assigning jobs to the CPU in intermittent manner, to decrease CPU power utilization, with static (offline) and online methods. In addition, [26] demonstrates an algorithm, nqPID, which claims to be more efficient than Pering’s AVG N algorithm, in both energy consumption and performance. A different approach is shown in [27], where energy optimization is achieved, by minimizing the leakage power drain (on the area of critical speed), which normally is increased through DVS methods. In paper [28], the authors propose static and dynamic algorithms for variable voltage scheduling, in real-time systems, like FPGAs and ASICs. The static algorithm creates a path analysis and task execution order, and then constructs a variable voltage schedule based on them. The dynamic one, also based on a static schedule, provides best-effort service to aperiodic tasks. Finally, [29], presents an overview of DVS methods and algorithms.

Moving to the area of applied methods, in real microprocessors, [30] demonstrates an application of the Dynamic Voltage and Frequency Management (DVFM), as it introduces DVFM in a microprocessor often used in embedded systems. As another example of application, in [31] we can see how techniques suggested by the authors were resolved in the IBM PowerPC 405LP processor. An extension to this paper is [32], where a dynamic-power-management architecture for embedded systems is presented, with focus on the previous processor.
IV. DIVING INTO DVS-DFS

A. Preliminaries

In order to be more ample in our effort to approach the various Dynamic Voltage Scheduling Techniques, we present here some important facts, which affect the way an electronic circuit works and responds. The processor of an embedded system, bases its functionality to millions of CMOS transistors, which constitute its core. From the literature, it can be found that the simple first order CMOS delay model, has energy consumption per sample, given by the formula:

\[ E(r) = CV_o^2 T_s f_{ref} \left[ \frac{V_o}{V_o} + \frac{r}{2} + \sqrt{\frac{V_o}{V_o} + \left(\frac{r}{2}\right)^2} \right] \]  

where \( C \) is the average switched capacitance per cycle, \( T_s \) is the sample period, \( f_{ref} \) is the operating frequency at \( V_{ref} \), \( r \) is the normalized processing rate i.e. \( r = \frac{f}{f_{ref}} \) and \( V_o = (V_{ref} - V_t)^2/V_{ref} \), with \( V_t \) being the threshold voltage. As we can see, the power dissipation in this kind of systems has a quadratic dependency on the operating voltage.

The supply voltage also determines the CMOS circuit delay \( \tau \) (\( \tau \) is proportional to \( V_{dd}/(V_{dd}-V_t)^2 \)), where \( V_{dd} \) is the supply voltage, and \( \alpha \) is a velocity saturation index. We can see that lowering the supply voltage, we increase the circuit delay and decrease the speed.

As we can understand from the previous relations, by varying the supply voltage of the circuit, we change drastically the energy consumption of the system. It also affects the frequency at which the system is working. So we have to have in mind that no change in voltage can happen alone, i.e. there will be a consequent change in the operating frequency too. On the other hand, if we want to change the frequency to a higher level, we need to increase the supplying voltage too. Of course, if we want to move to lower levels of speed, we can always reduce the voltage as well, and save energy.

On the next section we will present in more depth, six different DVS and DFS methods and applications, which rely on various field-areas of the PSL. These methods and applications are very well documented and supported in terms of literature. Each one is important to be demonstrated, to gain a more meaningful understanding of how the Dynamic Voltage Scheduling, the Dynamic Frequency Scheduling and as an extension, the Power Supply Level, can be used to address the energy consumption issue in embedded systems. We have to note though, that these techniques are just examples of the different ways to tackle the problem at hand, No one can argue that one method is better than the other, but only in terms of their effectiveness to reduce the power dissipation.

B. Real-Time DVS-DFS Algorithms

In this section, we will present real-time execution algorithms, introduced by Pillai and Shin, in [33]. We will address them in a theoretical point of view and then present some of the actual results taken by experiments on simulations and a real hardware platform.

First we describe the key concepts of the algorithms. We need to note that these algorithms integrate DVS-DFS mechanisms into the two most studied real-time schedulers, the Rate Monotonic (RM) and the Earliest Deadline First (EDF). For comprehension purposes, RM is a static priority scheduler, which assigns the actual priority of each task, depending on the period, i.e. it will always select the task which will need the shortest period of time for execution and of course is ready to run, whereas EDF will choose for execution, that task, that has the most impending deadline, thus making a dynamic priority scheduling. In the next paragraphs, we will use the following notations: \( T_i \) is an independent task that has an associated period \( P_i \) and a worst-case computation time \( C_i \).

1) Static Voltage Scaling: In this mechanism, the system chooses the lowest possible operating frequency (by scaling the frequency by a factor \( \alpha, 0 \leq \alpha \leq 1 \)), that will allow RM or EDF scheduler to suffice the deadlines, for a set of tasks assigned. So the frequency is statically set and can change, only when the task set is changed. The algorithm is shown in figure 1.

2) Cycle-Conserving RT-DVS: During the execution of real-time tasks, the worst-case computation requirements are used, but usually that’s not the case, i.e. tasks tend to use less than the worst-case time, on their calls. So, in this mechanism, every time a task completes, we compare the actual processor cycles that were allotted to the task, with the worst-case assignment of cycles. If there is a difference, implying idle cycles of the processor, the frequency and thus the voltage of the processor are reduced, and the power is conserved. This kind of algorithm is tightly-coupled with the operating system’s task manager, since it might demand reduction of operating frequency (and voltage) on a task completion, and increase upon release of a new one. We can see an implementation of this algorithm, in figures 2 and 3, for EDF and RM schedulers respectively.
In their hardware setup, they used an HP N3350 notebook computer, which had an AMD K6-2+ processor, with a maximum operating frequency 550MHz. Though a laptop is not the most ideal embedded system, since it is a more general purpose-applications oriented system, a verification of the efficiency of the algorithms, can be presented. Indeed, from the results taken, we can definitely argue that the simulations are supported, and with a very high degree of resemblance.

These results also give us a real estimation of the reduction of power consumption, varying in the range of 20% to 40%. The only deviation from the simulated results is the fact that now a real platform is used, which has a standard offset value in the power consumption, due to the different peripherals involved in the laptops configuration, i.e. display, co-processors and other components. These components are not included in the simulations, thus the deviation offset.

3) Look-Ahead RT-DVS: If we move to the other side of the table, we can argue that we can use the minimum operating frequency needed at the first stages of execution of a task, and increase it later on, if needed, but ensure that meeting of all deadlines is been achieved. That means we look ahead in the execution horizon, and we postpone the worst-case executions, by assigning the maximum work allowed in the slots allotted for each task, and thus we can execute at the current time being the minimum work needed to be done, to meet the deadline. This of course may require that higher operating frequencies will be needed later on, to meet this deadline. But if the task set, tends to use a smaller amount of execution time than the worst-case estimation, then the system could actually benefit from this technique. An implementation of this algorithm for EDF schedulers is shown in figure 5.

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).

In their hardware setup, they used an HP N3350 notebook computer, which had an AMD K6-2+ processor, with a maximum operating frequency 550MHz. Though a laptop is not the most ideal embedded system, since it is a more general purpose-applications oriented system, a verification of the efficiency of the algorithms, can be presented. Indeed, from the results taken, we can definitely argue that the simulations are supported, and with a very high degree of resemblance.

These results also give us a real estimation of the reduction of power consumption, varying in the range of 20% to 40%. The only deviation from the simulated results is the fact that now a real platform is used, which has a standard offset value in the power consumption, due to the different peripherals involved in the laptops configuration, i.e. display, co-processors and other components. These components are not included in the simulations, thus the deviation offset.

3) Look-Ahead RT-DVS: If we move to the other side of the table, we can argue that we can use the minimum operating frequency needed at the first stages of execution of a task, and increase it later on, if needed, but ensure that meeting of all deadlines is been achieved. That means we look ahead in the execution horizon, and we postpone the worst-case executions, by assigning the maximum work allowed in the slots allotted for each task, and thus we can execute at the current time being the minimum work needed to be done, to meet the deadline. This of course may require that higher operating frequencies will be needed later on, to meet this deadline. But if the task set, tends to use a smaller amount of execution time than the worst-case estimation, then the system could actually benefit from this technique. An implementation of this algorithm for EDF schedulers is shown in figure 5.

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).

In this paper, the authors present both simulation results and verification from a hardware platform implementation. For the simulation, they used programming language C++, and though with quite a few simplifications and assumptions, they managed to show that their algorithms do work, and actually improve the energy consumption factor. Their simulations included variation of the number of tasks, of the idle level of the processor, of the machine specification in terms of levels of frequency and respective voltage, and finally of the computation time needed. We can draw a general sketch of the various graphs presented, which demonstrates the improvement on the energy consumption (figure 4).
Figure 4: A general sketch of Energy (normalized) vs Utilization.

- line: EDF
- dashed line: Suggested Algorithms
- dotted line: Bound

(Trend summation of figures 9-12 in [33])

Figure 5: Look-Ahead DVS for EDF schedulers (figure 8 in [33])

Conclusions: (1) The very first invocation of a task may exceed the estimated-specified computation time boundary. This happens due to the phenomenon of “cold start”, where the processor of the operating system, demonstrates high rate of cache misses, page faults, TLB misses, etc. All these count towards the increase of the overhead of the system, i.e. delay execution of tasks. This problem is solved as soon as a few subsequent invocations are being made, and so the system moves to a “warm” state. (2) The attempt to dynamically add a task may cause sequential missed deadlines, of the already running tasks. This will be a particular problem especially in aggressive DVS algorithms, such as Look-Ahead EDF. So if this is to be avoided, the new task must be introduced immediately in the task set, so that the DVS algorithm bases its decisions on the new characteristics of the system, thus releasing the new task as soon as the old ones are finished, but having already estimated the parameters needed for its release.

C. DVS using Adaptive Filtering of Workload Traces

Here we present a different approach on DVS. By filtering a trace history, a workload profile is formed, and based on this, prediction is been done, which should minimize the energy consumption under given performance requirements. This new adaptive approach and the rest of this section are introduced and analyzed in [34].

The system model is shown in figure 6, in the form of a block diagram. The “task queue”, models the various tasks submitted for execution, like disk accesses, I/O requests, interrupts, etc. Each source of events-tasks, has an average rate of $\lambda_k$ ($k = 1, 2, \ldots, n$). An internal operating scheduler accepts all these tasks, and decides which one to push to the processor for execution. The average rate of task arrival in the processor, is $\lambda = \Sigma \lambda_k$. The processor measures the idle cycles during execution, and depending on the varying processing rate $\mu(r)$, it computes a normalized workload $w$. The Workload Monitor fixes the processing rate $\mu$, depending on the current $w$, but also on a history of workloads, from previous executions. It also feeds the DC/DC Converter, with the current value of $r$, so that the Converter decides the new $V(r)$. It also sets the operation of $f(r)$, fed to the processor.

Figure 6: Block diagram of a DVS processor system (figure 3 in [34])

Next we describe the Prediction Algorithm, as presented in the paper [34], and the assumptions – notations made and used:

- $\lambda_1$: Max task rate
- $\lambda_n$: Min task rate
- Select_frequency($x$):
  - Use lowest frequency $f_i \in \{f_1, \ldots, f_n \mid f_i < ... < f_n \}$ such that $x \leq (f_i / f_n)$
- Upon task_release($T_i$):
  - Set $c_{\text{left}} = C_i$;
  - Defer();
- Upon task_completion($T_i$):
  - Set $c_{\text{left}} = 0$;
  - Defer();
- During task_execution($T_i$):
  - Decrement $c_{\text{left}}$;
  - Defer();
  - Set $U = C_i / P_1 + \ldots + C_n / P_n$;
  - Set $s = 0$;
  - For $i = 1$ to $n$, $\{ T_i \mid T_i \in \{ T_1, \ldots, T_n \mid D_i \geq \ldots \geq D_n \} /\ast$note: reverse EDF order of tasks $\ast/$
  - Set $U = U - C_i / P_i$;
  - Set $x = \max(0, c_{\text{left}} - (1 - U)(D_i - D_n))$;
  - Set $U = U + (c_{\text{left}} - x)/(D_i - D_n)$;
  - Set $s = s + x$;
  - Select_frequency ($s/(D_n - \text{current\_time})$);
by the authors. If we let observation period be T, and w[n] the average normalized workload in the interval (n-1)T ≤ t ≤ nT. At the time t = nT, a decision must be made, for what processing rate to set for the next slot, that’s r[n+1]. This will be done based on the workload history. The prediction suggested by the authors, based on the previous notations and assumptions, is:

\[ w_p[n+1] = \sum_{k=0}^{N-1} h_n[k] w[n-k] \]  

(2)

where \( h_n[k] \) is an N-tap, adaptable FIR filter. This filter’s coefficients are revised in each examination interval, which is based on the error introduced by the difference between the processing rate (set using the workload prediction) and the actual value of the workload. Two more assumptions have been made:

1) Let \( L \), be the number of discrete processing levels available, such that:

\[ r \in R_L, R_L = \left\lfloor \frac{1}{L}, \frac{2}{L}, \ldots, 1 \right\rfloor \]  

(3)

using a uniform quantization interval, \( \Delta = 1/L \).

2) The minimum processing rate is 1/L.

Based on the previous workload prediction \( w_p[n+1] \), the processing rate \( r[n+1] \) is set such that

\[ r[n+1] = \left\lfloor \frac{w[n+1]}{\Delta} \right\rfloor \Delta \]  

(4)

In this paragraph we explain briefly the four different types of filters presented.

1) Moving Average Workload (MAW): In this filter, the average of the workload of the previous N Slots is calculated, and the filter predicts the next slot’s workload, based on this average, i.e. \( h_n[k] = 1/N \). As it is commented by the authors, and anyone can argue, this kind of filter is quite very simple, working as a low-pass filter, and thus improving the workload profile, by removing the potential noise existing in the traces history. But it will probably fail to respond correctly in workloads with time varying attributes.

2) Exponential Weighted Averaging (EWA): Here, each slot’s workload is taken into account, in decreasing exponential order, i.e. \( h_n[k] = a^k \), for all n, with a chosen \( a \), such that \( h_n[k] = 1 \), and \( a \) is positive. In more simple words, the previous slot gets the maximum weight, lesser weight on the slot before that, i.e. the second one, and so on.

3) Least Mean Square (LMS): The filter’s coefficients are modified based on the prediction error. If we let this error be \( w_e[n] = w[n] - w_p[n] \), where \( w[n] \) is the actual workload and \( w_p[n] \) the prediction workload from the previous slot, then we can arrive to the following rule, for this filter’s update of coefficients:

\[ h_{n+1}[k] = h_n[k] + \mu w_e[n] w[n-k] \]  

(5)

where \( \mu \) is the step size. The advantage of this kind of filter is that it is self-configured and thus it can learn from the workload history without us needing to worry about individual traces.

4) Expected Workload State (EWS): Here, no filtering technique is used. Using the quantization presented before in equation (3) and by letting \( P = [p_j], 0 \leq i \leq L, 0 \leq j \leq L \), we can form a square matrix with elements \( p_{ij} \), so that:

\[ p_{ij} = \text{Prob}\{w[r+1] = w_j | w[r] = w_i\} \]

where \( w_k \) is the \( k \)th workload out of \( L+1 \) levels. So the workload is predicted as:

\[ w_p[n+1] = E\{w_p[n+1]\} = \sum_{j=0}^{L} w_j p_{ij} \]  

(6)

In this way, the matrix is being updated in every slot, in every actual state transition. So the \((r+1)^{th}\) state can depend on the previous N states.

Using the four previously presented filters and methods, a comparison on the level of Root-Mean-Square error, made by the authors, lead to the conclusion, that LMS technique, outperforms the other ones, and produces the best results with \( N=3 \) taps.

Another important note pointed out in this paper is the latency overhead involved for the processing rate updates. This actually exists due to the limited switching capability of the voltage regulator, which though it can go down to a few tens of a microsecond, it still is enough latency to produce a lock. That is why voltage and operating frequency should not be changed in parallel, but with a more specific order. In fact, when we want to reduce voltage and frequency to a lower state, we first lower the frequency and then the voltage level, whereas if we want to move to an increased level of frequency and voltage operation, we first move the voltage and then the frequency update proceeds. This procedure ensures that the processor has always at least the right level of voltage to operate at the given clock, and thus we avoid data corruption from circuit failures.

If we want to comment and compare results, on the energy consumed using the DVS and without using the DVS, we can define the Energy Savings Ratio, which is the ratio of the energy consumption without DVS and the energy
consumption with DVS. Maximum savings can be considered to occur when we set the processing rate equal to the average workload over the entire period of testing. Also we can define the “perfect” ESR, which is the case of perfect prediction for the next workload. An overview graph of the effect of discrete processing levels L, to the Energy Savings Ratio (ESR) is given in figure 7, where LMS filter is used out of the four presented.

As a conclusion from this paper, we can stress that maximum energy savings can be achieved, if the processing rate is set to the overall average workload. But this is most of the times impossible to know in advance, and even if somehow possible, it can lead to high performance penalties. That is why with frequent processing rate updates we can secure that the performance penalty will be low, or even limited to the minimum possible.

D. Parametric Intra-Task DVS

So far we have examined the potential of exploiting the DVS techniques, by varying the level of operating voltage and frequency, just before the next task executes, based upon the difference between the next task’s worst execution time (WCET) estimate and its deadline, adding any time gaps created from the execution of previous tasks, where they completed before the WCET estimation. These methods belong to the inter-DVS. There is also the intra-DVS area, where the voltage and / or frequency can be set at various levels, within the execution of a task, based upon the progress of the execution, with reference to a given, pre-determined path. That is, at each of the available levels of operation, a comparison is being made between the already estimated and remaining worst case execution cycles and the actual worst case execution cycles. If the state of the system produces a faster execution than the estimated one, then voltage and / or frequency can scale down, proportionally to the difference, to save excess power dissipation. If on the other hand, the system moves slower than what is expected, so that the deadline meets, a scale up on voltage is carried on, so that the task stays in timeline limit.

A general benefit of this family of DVS techniques is that the variations on the operating voltage and / or frequency of the processor, do not involve the OS scheduler, and thus are independent and more flexible (this is not the case on the first paper’s methods we presented in section B, where those techniques’ efficiency and performance, depend on the actual implementation of the scheduler).

In this paper [35], Walsh et al. elaborate on the intra-DVS area and present a parametric method which scales voltage / frequency depending on the number of remaining worst case execution cycles (RWEC).

Moving to an analysis of the Intra-VS methods, we can mention more details, on how an actual transformation of the code of a task is being done, to implement this kind of method. We consider having a block of code, and the RWEC known. In this case, if the code has if-statements and loop exit edges, then voltage scaling code (VSC) is inserted at those points. To be more precise, this peace of code (VSC) is placed right after the least costly branch, if the sum of this branch and the VSC overhead do not exceed the most costly branch. This is called B-Type DVS. There is also the L-Type DVS, where we place the VSC right after the loop exits, if only the processor will be able to execute the specific task in the specified deadline, including the additional VSC overhead.

If we consider the case of an L-Type DVS, we will have a Control Flow Graph (CFG), similar to figure 8. Here we need to point out that in order to estimate the RWEC statically, we need to have a bound on the maximum number of iterations to be executed by the loop, either from an analysis of the program-task, or by user submission of the RWEC. For example, let’s assume that a program’s code includes a loop which has a bound of N iterations, with each iteration demanding c₁ number of cycles, and at this point, c₉ cycles are left for the completion of the task. Assuming we do not know the total number of iterations, i.e. it is a dynamic parameter, then we have to assume that the iterations left are the maximum number, i.e. N. In either case the parameter n, which is the actual run time iterations, is n ≤ N. In this case we consider no scaling of the voltage. So the energy consumption can be estimated by:

\[
E_{NonScaled} \approx V^2(n_c L + c_R)
\]  

(7)

By applying the L-Type DVS, at the exit point of the loop, we can reduce the energy consumption, because an adjustment of the voltage will be done for the rest of the code of the task. The authors give the next function, for an estimation of the energy consumption. Nevertheless, no proof is provided. Also
the extra cycles introduced by the L-Type DVS are not included:

\[ E_{NonScaled} \approx V^2 (nc_L + \frac{c_R}{(N-n)c_L + c_R})^2 c_R \]  \hspace{1cm} (8)

Moving a step further, by applying this DVS method, at the start of the loop, we can reach even better results, by introducing the parameter \( n \) in the estimation:

\[ E_{NonScaled} \approx (V \frac{nc_L + c_R}{Nc_L + c_R})^2 (nc_L + c_R) \]  \hspace{1cm} (9)

The authors name the third type of DVS method, the P-Type (Parametric) DVS. By elaborating on the three estimations presented, we can decide which the best method for reducing the energy consumption is. An important note to be made is that the improvement of the energy consumption, by using L-Type and P-Type DVS, heavily depends on the proportion of loop code in the overall program. The paper presents two graphs, in which we can see that if the whole program code, is only the loop itself, then the improvement on energy consumption, and especially using the P-Type DVS, is much more evident than in the case where the loop accounts for only one tenth of the overall code.

To demonstrate the structure of a task’s code, that uses P-Type DVS, we draw a CFG, like the one in figure 9. As we can observe, the insertion point of the P-Type DVS code, is at the very early beginning of the execution. This of course poses the limitation that the values of the boundaries of the loops must be known before the execution; this can be done either by passing them as arguments - parameters during the task’s call, or by some other calculations at the very beginning of the task, and before the P-Type code. The overhead of the VSC mustn’t be excessive, i.e. the function used to calculate the operation voltage level \( V \), can be linear or polynomial, but in

any case, there must be a tradeoff between time to execute and accuracy needed.

A combination of B-Type and P-Type DVS is possible and also available in the paper. We need to be aware of the points mentioned before, i.e. where to place a B-Type and where a P-Type DVS VSC, by taking into account the branch costs and the extra cost added by the VSC overhead and also the actual estimation of the RWEC.

Finally, we can construct a general parametric IntraVS algorithm, which combines the knowledge and advantages of the above presented methods. The algorithm is shown in figure 11. We rephrased and combined the various steps, to make the algorithm more comprehensive and compact.

We can’t stress enough the importance of this kind of techniques. Their significance lies on the fact that intra-DVS techniques change the operating state of the processor while executing a task and not between tasks. So if a task takes a
long time to execute, e.g. due to some loops, we can improve the energy consumption, i.e. reduce it, by varying the speed – performance of the CPU, depending on the RWEC. Of course, we always need to have in mind the tradeoff between VSC overhead execution time and actual reduction of the energy consumption.

In the previously presented paper, the algorithms described are elaborating on the internal of the execution of a task, by moving the operating state of the system. An interesting aspect brought up by the authors of [37], is that during the execution of a task (#1), some other tasks may arise, which are part of the first one, and might or might not have overlapping execution with task #1. Let us consider figure 12.(a). We can see a sequence of tasks, which overlap in execution. Also in figure 12(b), a potential overlapping of deadlines is shown. A combined problem arises here: what operating state to choose, at each moment, which will satisfy all the tasks to meet their deadlines, but at the same time consume the least of energy?

In this paper ([37]), the authors work again on intra-level DVS methods, but introduce the idea of using checkpoints during the execution of the several stages of the program. In this way, for each transition from one checkpoint to the next, we can calculate the frequency limit of the system, depending on the available power to dissipate, and the maximum power to be dedicated per cycle. Using the value of frequency limit, we can calculate the optimal frequency that can be used at the current stage, which will consume the least power possible. This can be better understood by studying the example given in figure 13, where a simple piece of code is given, with various structural blocks, and checkpoints at the crucial parts of the code.

**Full parametric IntraVS Algorithm:**

1. Generate the CFG of the task.
2. Static WCET and BCET analysis on the graph’s basic blocks, based on Healy et al. [36].
3. Determination of the loops’ space size, using parametric WCET analysis.
4. Unfolding of the graph, to exclude cyclic paths, preserving information about the loops’ structure.
5. Moving in a bottom-up approach of the above unfolding, generate the loops’ cost index table. The table includes: loop’s index, the current cycle cost (evaluated using the max number indicated by the user and a parametric formula), and the cost parametric formula.
6. When all dependent variables are known, update the loops’ index table and introduce V/FSC (i.e. P-Type DVS).
7. If there are if-statement branches that qualify for, introduce B-Type DVS V/FSC after the branches.

**Figure 10: Combined P-Type and L-Type DVS (figure 4, in [35])**

**Figure 11: The steps of the placement algorithm**

**E. Profile-Based DVS using Program Checkpoints**

In order to apply the idea of DVS using the checkpoints, we need an algorithm to calculate the frequency limit, the optimal frequency and the corresponding values of optimal voltage, depending on various situations that may arise. This algorithm (#1) is demonstrated in simple steps in figure 14. As we can see, in the first step, the algorithm creates a list of all possible events that might take place, from the current checkpoint and on, till the uttermost deadline of all the potential checkpoint transitions. At the next step, the frequency limit is being
calculated. In order to do that, we take into account the values from the checkpoint database (CDB) table (figure 13(b)), which has entries for each checkpoint transition’s acceptable lower and upper bounds. We also need to consider the checkpoint power profile database (CPDB) table, which provides us with the minimum and maximum energy / power dissipated and cycle count for each checkpoint transition. Proceeding in the algorithm, we calculate the optimal frequency, which is the frequency the system should use, to run and thus meet the time and energy deadlines. The frequency limit calculated before, is used to calculate the range of possible levels of frequencies which the CDB allows. From this range, an optimal selection is made for the operating state.

\[
\text{\( f_{\text{limit}} = \sqrt{\frac{power\_limit \times \text{freq}^2}{\text{max\_power\_per\_cycle}} \)}
\]

\[
\text{power\_limit = maximum dissipated power derived from run-time power constraint}
\]

<table>
<thead>
<tr>
<th>CDB</th>
<th>Checkpoint Transition</th>
<th>Min Time (ms)</th>
<th>Max Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-2</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>2-3</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3-3</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>3-4</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

(a)          
(b)          

Figure 13: Code example with program checkpoints, imposing time constraints (figure 2 in [37]).

Based on the previous algorithm (#1), the authors of [37] present a Slack-based algorithm (#2), which is using algorithm #1, but also exploits some other attributes of the code of a program, and helps the system become even more power efficient. In this algorithm (#2, figure 15), program checkpoints are initially introduced at the various key points of the code, i.e. loops, branches and function calls. In fact, if the point is a function call, the node is called funct, if it is a loop header, loop, and if it is a branch or any other kind, it is called normal. A preliminary phase, estimates the minimum and maximum number of execution cycles, for each of these nodes, and the maximum number of iterations for each loop. Thus a CPDB table is constructed and filled, at each node. A new CDB is built for every time a checkpoint is encountered, and is used as input to the previously analyzed algorithm (#1). In the preliminary phase, a hierarchical control flow graph (HCFG) has been created. This kind of graph represents the program in the form of nodes, where nodes, are all checkpoints introduced in the code. In this kind of graph you can also save information related to each of the nodes.

Figure 14: DVS Algorithm using program checkpoints (based on figure 3 in [37]).

If we analyze further the behavior of the algorithm #2, when it encounters one of the types of nodes, we can mention the following: 1) Normal Nodes: two Formulas are used to calculate the cycle count. Formula 1 uses the length of the longest path from the current node to the end point of the checkpoint transition. Formula 2 keeps track of the cycles needed since the start of execution of the current checkpoint. The algorithm actually uses as best estimation, the minimum of the two Formulas, as the cycle count. 2) Function Call or
executions of tasks. The benchmark "deciding the updated values of frequency and voltage, were overhead, nor the time and power overhead of the code, for deciding the updated values of frequency and voltage, were taken into account, the simulations done were close to real executions of tasks. The benchmark "paraffins" was used for the comparisons. The results are quite impressive. The Formula 1, Formula 2 and the minimum of the two, were tested, to find the most efficient in terms of frequency limit and thus power consumption. Using the Formula 2, was more efficient by 63% than Formula 1, and finally using the minimum of the two, was overall 82% more efficient than the original execution without the application of the slack-based DVS algorithm.

Of course the algorithm will need improvements. One of the more crucial ones is for the algorithm to take into account the power and time overhead. Time overhead is important to be considered, because, frequency and voltage scaling procedures do take time execution, which is an overhead, and could grow into a significant problem if too many VSC are introduced, to become a major portion of the program’s code.

F. Battery-aware DVS in Multiprocessor Embedded System

So far we have examined DVS algorithms that can be used to improve the power consumption of an embedded system. These methods, address the energy, through the variation of the voltage and frequency level. In the following paragraphs, we will describe a different approach to the problem. This technique, which emphasizes on the load current profile (LCP) of the battery used to support and supply the system, is oriented towards multiprocessor embedded systems architectures. That is, it considers the current drawn from the battery, for the execution of each individual task, with the latest finish times in the schedule, but taking into account the number of processors in the system. This technique is described in [38], where the authors present the DVS algorithm and some experimental results to support their case.

The proposed Battery-Aware DVS algorithm has the following key points: first, partition a given task schedule (which describes what task will be executed by what processor and at what time), into discrete steps, using the procedure outlined in figure 16. Then, assuming that the task execution order is fixed, and the current of each task is constant, and using the algorithm shown in figure 17, we get as output the supply voltage of all the tasks, during each step of the task schedule. The general idea of the algorithm is to spread the execution of a task or tasks, as long as the deadline of the task allows, so that the current used to execute them, will be reduced as much as possible. The innovation of the algorithm, though, is not this. Is the fact that it repeats itself, until either all the slack available by the early execution of tasks is used up, or the supply voltages of all the PE’s in every step reach the lowest level possible.

---

**Loop Headers:** Here, the tightest time constraint of the current path must be determined (explanation in figure 15).

1. At a checkpoint execution, create a new CDB with the active checkpoint transitions.
2. Check type of current node in the HCFG.
   - **Case 1:** Node type is normal (not function call or loop header).
     a. Estimate the number of cycles \( C \) from curr. node to all active checkpoint transitions.
     \[
     \text{Formula 1:}
     C = \text{longest_path}(\text{curr node, active checkpoint transition end point})
     \]
     Or for inherited time constraints:
     \[
     C = \text{longest_path_length}(\text{curr node, end node of curr sub-CFG}) + \text{extraC}.
     \]
     Where extraC is obtained by
     \[
     \text{extraC} = \text{remaining number of iterations} \times \text{max cycle per iter} + \text{inherited max cycle (for parent node of type loop header)}
     \]
     or \( \text{extraC} = \text{inherited max cycle (for parent node of type function)} \)
   - **Formula 2:**
     \[
     C = \text{max profiled number of cycles for active checkpoint transition} - \text{elapsed number of cycles for active checkpoint transition}
     \]
   b. Select for each checkpoint transition\( \text{C = CFormula1 or C = CFormula2 or C = min(CFormula1,CFormula2)} \)
   c. Create a new CPDB using C for each active checkpoint transition.
   d. Invoke DVS algorithm #1, using new CDB and CPDB.
   e. Continue with program execution.
   - **Case 2:** Node is function call or loop header.
     a. Calculate the tightest active checkpoint transition selecting by frequency
     \[
     \text{Formula 2:}
     C = \text{min} (CFormula1, CFormula2)
     \]
     Where extraC is obtained by
     \[
     \text{extraC} = \text{remaining number of iterations} \times \text{max cycle per iter} + \text{inherited max cycle (for parent node of type loop header)}
     \]
     or \( \text{extraC} = \text{inherited max cycle (for parent node of type function)} \)
     b. Add an end-node to the curr node’s sub-CFG with max cycle = \( C \).
     c. Add inherited time constraint to CDB with time = inherited constraint remaining time.
     d. Continue with program execution.

---

**Figure 15:** Slack-Based DVS Algorithm (#2), using program checkpoints (figure 6 in [37])

We need to point out that the creation of a CFG, finding reachable nodes and the longest path in a HCFG, can be pre-computed and thus the run-time processing is spent on updating the CDB or CPDB tables, HCFG, and also estimating the cycle count.

The algorithm #2, which basically is an improved version of #1, was tested, thoroughly, using the COPPER simulation framework. Although no frequency and voltage scaling time overhead, nor the time and power overhead of the code, for deciding the updated values of frequency and voltage, were taken into account, the simulations done were close to real executions of tasks. The benchmark "paraffins" was used for the comparisons. The results are quite impressive. The Formula 1, Formula 2 and the minimum of the two, were tested, to find the most efficient in terms of frequency limit and thus power consumption. Using the Formula 2, was more efficient by 63% than Formula 1, and finally using the minimum of the two, was overall 82% more efficient than the original execution without the application of the slack-based DVS algorithm.

**Input:** task schedule

1. Begin from the start time of the first task.
2. When the start time of any task or the finish time of any task is met, mark this time point, as the end of the current step and the beginning of the next step.
3. The duration of a step represents its length.
4. The current of a step, is the sum of the currents of the tasks contained in this step.
5. The length and current of all the steps constitute the LCP for the battery model.

**Output:** partitioned in steps, task schedule

**Figure 16:** Procedure for partitioning a task schedule.
Input: partitioned in steps, task schedule

1 ; /*length of task */
sl = true; /* state where slack is used up */
spl = true; /* supply voltage of all the tasks inside each */
* step reaches the lowest level */
while(true) {
\Delta \sigma_{max} = 0;
k = 0;
store the length and current of all the steps;
store the voltage of all the tasks inside each step;
for each step, {
\text{for each PE,}} \\ 
\text{reduce } V_j \text{ by } dV_j \text{ such that } s_i = (V_j - dV_j) / V_j; \\
\text{compute current;}
compute cost function reduction \Delta \sigma;
\text{if (} \Delta \sigma_{max} < \Delta \sigma \text{)} \\ 
\Delta \sigma_{max} = \Delta \sigma;
k = k;
set l and current of step, to its old value;
set the voltage of all the tasks inside step, to their old values;
}
\text{scale step, with scaling factor } s_i; \\
\text{if (} sl || spl \text{) break;}
}
Output: The supply voltage of all the tasks during each step

Figure 17: The proposed DVS algorithm

Here we need to comment on the complexity of the algorithm. Let us assume there are n total steps. Let’s also define the relation of dV and the amount of slack produced, by a factor m, where for smaller values of dV and larger values of slack, m will be larger. Taking into account these, we can argue that the complexity of the algorithm has an upper bound of O(mn^2). This introduces an increase in the computation time needed from this algorithm, and it is a disadvantage. But as the authors argue, the results of the simulations are very encouraging, and the algorithm sounds very promising. Next we present a sketch that can be drawn using the data from Table II in [38], where it presents the Voltage Scaling Results, from experiments done. The table has entries for 25 random task graphs, and the corresponding number of nodes and edges per task and number of processing elements per task. It also shows the cost function reduction introduced and the respective CPU time. The cost function reduction is a percentage ratio of the battery charge consumption with and without DVS method.

As we can see from the previous sketch, the percent reduction of the cost function is in every randomly chosen task graph higher than zero, which means that the algorithm works well. We need to note that the RAC, appearing in the last place of the random graphs axis, is a robot arm controller real task graph. This shows that even in a real time system, with actual realistic graph, the algorithm proves robust. Also, as the authors pointed out, we need to mention that from the results of the Table II in [38], reduction cost function seems to depend on the slack and the amount of concurrent processing in the original schedule. That means it does not directly depend only on the size of the task graph. This can be seen from the sketch as well, where the percentage of reduction varies, and does not show a trend with respect to the size of the graph.

Figure 18: A sketch of the percentage reduction on battery charge consumption, based on the Table II, in [38].

G. DVS Technique for Low-Power Multimedia Applications Using Buffers

In this section, we describe the idea presented in [39], where the authors proceed into introducing task buffers in the embedded system’s architecture, so that the available slack time is fully exploit by buffering multiple input data, so that there is always at least one application that can be running on the processor. The main focus of this approach, as the title depicts, is soft real-time applications, like multimedia applications, where latency can be tolerated to some factor.

An example of the suggested approach is presented in figure 19. In part (a), we can see a normal execution of a series of tasks, which have a run time and an idle time, due to the difference between worst-case execution time and the actual execution time. If we buffer the tasks, and also modify the operating voltage they use, we can exploit the idle times, and also stretch the execution of each, so that though they will take longer to execute, since voltage and thus frequency will decrease, they will be executed using a smaller amount of energy (part (b)).

In order to establish a clearer site of the algorithm, the authors use the following notations:

\( S_j \): the \( j \)th schedule instance of a given task set ( \( j \geq 1 \))
\( \text{WST}_j \): worst-case slack time of \( S_j \)
\( \text{VST}_j \): workload-variation slack time of \( S_j \)
\( \text{WET}_j \): worst-case execution time of \( S_j \) at full supply voltage
BET$_j$: best-case execution time of $s_j$ at full supply voltage
AET$_j$: actual execution time of $s_j$
OP$_j$: occupation period of $s_j$. It’s the max time that this task can run without violating any time constraints.

These notations are also illustrated in figure 20, where the algorithm is presented.

Moving to the buffer size estimation problem, three cases are examined here: single task problem, single-task that consists of multiple subtasks with different parameters and multi-task. For overview purposes, we will just present the results found by the authors, excluding the proofs. For the first case problem, we can state that the minimum size of the task buffer, $h_{min}$ is given by:

$$h_{min} = \left\lceil \frac{VST}{T} \right\rceil = \left\lceil \frac{WET}{BET} - 1 \right\rceil$$  \hspace{1cm} (10)$$

For the second case problem, each subtask has a different VST, so in order to find the minimum size of buffer, we need to find the VST value for each subtask, and then take the maximum of this set of values, as the $h_{min}$:

$$VST_{ij} = \frac{NT}{\sum_{BET} (WET_{j+1} - BET_{j+1})}$$  \hspace{1cm} (11)$$

where $NT = \sum T_i$

and using (11) => $h_j = \left\lceil \frac{VST_{ij}}{T} \right\rceil$  \hspace{1cm} (12)$$

So $h_{min} = \max \{h_j\}$. \hspace{1cm} (13)$$

Moving to the third case, and by defining $H$ as the hyper period of the given task set (that is the least common multiple of tasks’ period), VST value for each task is given by:

$$VST_j = \frac{H}{\sum_{BET} (WET_{j+1} - BET_{j+1})}$$  \hspace{1cm} (14)$$

and $h_{min} = \max \{h_j\}$ \hspace{1cm} (15)$$

Using these symbol-entities, we can now give a description of the algorithm, which is shown in figure 20. We need to state the assumptions made by the authors. For all $j$, the WET$_j$ and BET$_j$ are known or approximately estimated. Of course this is not always the case, and an improvement must be made, as the authors themselves point out in the end–conclusion of the paper. Also they assume that a fixed-priority scheduling algorithm already supplies the algorithm with a task set.

For verification purposes, the algorithm was tested on a platform that had an UltraSparc-II CPU, but implemented the ARM® microprocessor’s power–delay curve. A video phone application was used, which was composed of an MPEG-4 video encoder / decoder and a VSELP (Vector-Sum-Exited Leaner Prediction) voice encoder / decoder.

From the results presented, we can compare various modes of operation of the platform. These modes are: no VS technique with power-down, intra-task timeslot VS technique, the proposed buffering technique, the proposed buffering technique, but with multiple subtasks, for best execution time assumptions and average execution time assumptions. The proposed algorithm achieves the best energy consumption reduction among the other techniques. However, the amount of energy saving varies, with the application execution time.

What we can also comment and it is very important for the proposed approach, is the effect of the number of buffers (size), to the energy dissipation, due to latency constraints. Here we have to state that the size of buffer needed, depends clearly on the application to be applied. In order to illustrate the relation of the size of the buffer to the latency constraints, we redraw a sketch of the graph in figure 9, shown in paper [38]. In this sketch (figure 21), the effect of latency is demonstrated, in energy saving, using the MPEG-4 video decoder, which needs maximum 3 size buffer for best results, and no latency. As it is expected, the more latency we impose in the implementation, by reducing the size of the buffer, the less energy saving we achieve. Another important conclusion is that by applying just one extra buffer, we achieve the most reduction in energy dissipation. So it is rather easy to implement smart buffers that will minimize the energy consumption, without estimating the best number of buffers,
which might be a difficult or impossible problem to solve and will consume computation time, with rather little extra effect to the whole.

Through the previous analysis of the various methods and techniques, we can pick out some interesting points, which need our attention, when designing a DVS or DFS, or even the architecture of a more general computing system.

1. Phenomenon of “cold start”: at the very first invocation of a task, it may exceed the estimated or specified computation time boundary. This happens due to the phenomenon of “cold start”, where the processor of the operating system, demonstrates high rate of cache misses, page faults, TLB misses, etc. All these count towards the increase of the overhead of the system, i.e. delay execution of tasks. This problem is solved as soon as a few subsequent invocations are being made, and so the system moves to a “warm” state.

2. The attempt to dynamically add a task may cause sequential missed deadlines, of the already running tasks, especially in aggressive DVS algorithms. To avoid this, the new task must be introduced immediately in the task set, so that the DVS algorithm bases its decisions on the new characteristics of the system, thus releasing the new task as soon as the old ones are finished, but having already estimated the parameters needed for its release.

3. Latency overhead and changing voltage and frequency levels: This problem actually exists due to the limited switching capability of the voltage regulator (used to trim the operating voltage level of the processor), which though it can go down to a few tens of a microsecond, it still is enough latency to produce a lock. That is why voltage and operating frequency should not be changed in parallel, but with a more specific order. In fact, when we want to reduce voltage and frequency to a lower state, we first lower the frequency and then the voltage level, whereas if we want to move to an increased level of frequency and voltage operation, we first move the voltage and then the frequency update proceeds. This procedure ensures that the processor has always at least the right level of voltage to operate at the given clock, and thus we avoid data corruption from circuit failures.

4. VSC overhead: In methods where overhead code is introduced, for checkpoints etc, we always need to have in mind the tradeoff between VSC overhead execution time and

![Figure 20: Proposed technique, including the notations used (figure 2 and 3 in [39])](image)

![Figure 21: The effect of latency constraints in saving energy. (a) no constraints (extra buffer size = 3), (b) extra buffer size = 2, (c) extra buffer size = 1, (d) no input buffering (extra buffer size = 0) (based on figure 9 in [39]).](image)
actual reduction of the energy consumption we achieve by introducing the overhead.

5. Sometimes, a small improvement on the way a system works could reduce by a great factor, the energy consumption of the system. A simple yet characteristic example is the idea of using buffers to feed the processor with tasks. As we discovered, by applying just one extra buffer, the reduction of energy dissipation maximizes. It is easy to implement smart buffers that minimize the energy dissipation, without the need to estimate the best number of buffers. This might be a difficult or impossible task to solve and will probably consume computation time, with rather little extra effect to the whole effort for reducing energy consumption.

It is clear that in the future, faster and smarter computers will be deployed, with more functionality and computational power. The only problem that will still be present is the power consumption of such a system. In that sense, more efforts and research ideas are expected from the academic community and the industrial world, to tackle this crucial problem of energy dissipation.

REFERENCES


[38] Yuan Cai, Sudhakar M.Reddy, Irith Pomeranz, Bashir M Al-Hashimi, “Battery-aware Dynamic Voltage Scaling in Multiprocessor Embedded System”, in …

[39] Chaeseok Im, Huiseok Kim, Soonhoi Ha, “Dynamic Voltage Scheduling Technique for Low-Power Multimedia Applications Using Buffers”, in ISLPED ’01, August 6-7, 2001, Huntington Beach, California, USA.