UNIVERSITY
OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Sofia Mäkikyrö

APPLICATION ANALYSES OF ULTRA-LOW-ENERGY PROCESSOR

Master's Thesis
Master's Degree Programme in Biomedical Engineering
November 2020

Mäkikyrö S. (2020) Application analyses of ultra-low-energy processor. University of Oulu, Master's Degree Programme in Biomedical Engineering. Master's Thesis, 65 p.

ABSTRACT

Low energy consumption has become a critical design feature in modern systems. Internet of Things, wearables and other portable devices create increasing demand for low power design where device size is dictated by battery and low energy means longer battery life and smaller physical size. These are crucial features for wearables and especially implantable medical devices.

There are several low power and energy efficient techniques which are applied at different abstraction levels of the system design. A technique usually utilizing software control and hardware features is DVFS (dynamic voltage and frequency scaling), a dynamic power management technique which decreases processor clock frequency and supply voltage. Reduction in energy consumption is achieved with the cost of reduced performance. One of the questions with DVFS is how the execution frequencies are defined.

This thesis presents a method for frequency optimization for applications executed on a single core processor. Execution trace data is used to profile the application. FreeRTOS operating system is used although tracing can be implemented with any real-time operating system executing tasks as separate threads. Based on profiling and user-defined data, task execution frequencies are defined assuming that execution time scales linearly with the frequency.

A near-threshold ARM Cortex M3 with integrated power management and phase-locked loop is used for measurements. The measurements show that energy savings can be achieved without affecting correct application execution. However, the reduction in energy consumption depends highly on the system used and the application execution profile. Iterative testing and frequency optimization are required to ensure adequate performance. For energy efficiency optimization, energy consumption needs to be considered in every phase of the design.

Keywords: dynamic voltage and frequency scaling, energy efficiency, embedded systems, low power design

Mäkikyrö S. (2020) Matalan energiankulutuksen prosessorin sovellusanalyysi. Oulun yliopisto, lääketieteen tekniikan maisteriohjelma. Diplomityö, 65 s.

TIIVISTELMÄ

Matala energiankulutus on keskeinen ominaisuus nykyisten järjestelmien suunnittelussa. Esineiden Internet ja puettava tietotekniikka luovat tarpeen yhä pienemmälle energiankulutukselle. Laitteen koko määräytyy akun koon mukana. Matala tehonkulutus tarkoittaa pidempää akunkestoa ja pienempää fyysista kokoa. Nämä ovat ratkaisevia ominaisuuksia, erityisesti implantoitaville lääkinnällisille laitteille.

Energiatehokkuuteen ja matalaan energiankulutukseen tähtääviä menetelmiä voidaan soveltaa eri abstraktiotasoilla järjestelmän suunnittelussa. Dynaaminen jännitteen ja taajuuden skaalaus on menetelmä, millä pyritään alentamaan dynaamista tehonkulutusta säätelemällä käyttöjännitettä ja kellotaajuutta. Suorituskyvyn kustannuksella on mahdollista saavuttaa matalampi energiankulutus. Keskeinen kysymys on, miten käytettävät kellotaajuudet tulee määritellä.

Tässä diplomityössä kehitetään menetelmä, jota voidaan käyttää optimaalisten kellotaajuuksien määrittämiseen. Suorituksen aikana kerättävää dataa käytetään ohjelman profilointiin ja optimointimallin luomiseen. Suoritusdatan kerääminen on kehitetty FreeRTOS-käyttöjärjestelmälle, mutta periaate on sovellettavissa käyttöjärjestelmille, joissa tehtävät suoritetaan erillisissä prosesseissa. Profilointidata hyödynnetään yhdessä käyttäjän syöttämän data kanssa kellotaajuuksien määrittämiseen olettaen, että suoritusaika skaalautuu lineaarisesti kellotaajuden kanssa. Suositustaajuudet määritetään jokaiselle prosessille erikseen.

Mittauksissa käytettiin ARM Cortex M3 prosessoria integroidulla tehonhallinnalla ja vaihelukolla. Mittaustulokset osoittavat, että energiankulutusta voidaan pienentää vaikuttamatta sovelluksen virheettömään suoritukseen. Saavutettava hyöty tehonkulutuksessa on riippuvainen käytettävästä järjestelmästä ja sovelluksen suoritusprofiilista. Riittävä suorituskyky täytyy varmistaa iteratiivisella testaamisella ja kellotaajuuksien optimoinnilla. Tehonkulutus ja energiatehokkuus täytyy huomioida suunnitteluprosessin jokaisella osa-alueella, jotta parhaat tulokset saavutetaan.

Avainsanat: dynaaminen jännitteen ja taajuuden skaalaus, energiatehokkuus, sulautetut järjestelmät, matalan tehonkulutuksen järjestelmäsuunnittelu

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
ABBREVIATIONS
1. INTRODUCTION 8
2. DEVELOPMENT ISSUES 11
 2.1. Energy consumption 11
 2.2. Timing challenges 11
  2.2.1. Power modes 12
  2.2.2. DVFS 13
 2.3. Reliability 14
 2.4. Benchmarking 14
3. TECHNIQUES FOR LOW ENERGY CONSUMPTION 16
 3.1. Hardware techniques 16
 3.2. Algorithm level techniques 17
 3.3. Scheduling 19
 3.4. DVFS algorithms 20
 3.5. The role of software 24
4. PROFILING FOR THE NEAR-THRESHOLD PROCESSOR 25
 4.1. Task execution 26
 4.2. DVFS implementation 27
 4.3. The optimization model 28
 4.4. Software execution tracing 29
 4.5. Profiling 31
  4.5.1. Profiling data visualization 32
  4.5.2. Target system 33
  4.5.3. User-defined data 36
5. MEASUREMENTS 38
 5.1. Tracing and measurement set-up 38
 5.2. Test application 38
 5.3. Frequency optimization 39
  5.3.1. Trace 1 39
  5.3.2. Trace 2 41
  5.3.3. Trace 3 42
  5.3.4. Trace 4 43
 5.4. Optimization results 44
  5.4.1. Power measurement results 45
  5.4.2. Optimization discussion 46
6. DISCUSSION 48
 6.1. Limitations 48
 6.2. Use considerations and improvements 50
  6.2.1. Tasks suitable for optimization 50
  6.2.2. Required slack time 50
  6.2.3. Task execution, deadline misses and prioritization 51
  6.2.4. Additional profiling and detailed modelling 52

6.2.5. Relation between execution frequency and execution time 52
6.3. Conclusion 53
7. SUMMARY 54
8. REFERENCES 55
APPENDICES

FOREWORD

I would like to thank everyone involved in the process of writing this thesis.

Oulu, 4.11.2020

Sofia Mäkikyrö

ABBREVIATIONS

C_L	load capacitance
c	task instance count
CMOS	complementary metal oxide semiconductor
CPU	central processing unit
DVFS	dynamic voltage and frequency scaling
E	energy per cycle
E_{avg}	average energy consumption
EDF	earliest deadline first
f	execution frequency
f_{scaled}	optimized execution frequency
GPOS	general purpose operating system
I_s	capacitive current
I/O	input/output
KWS	Keyword spotting
OP	operating point, a voltage-frequency pair
OS	operating system
P	power
PLL	phase locked loop
RM	rate monotonic scheduling
RTOS	real-time operating system
RTT	Real-Time Transfer
s	scaling factor
SWD	Serial Wire Debug
t	time
T	charging interval
V_{dd}	supply voltage
V_{out}	output voltage
wcec	the largest execution cycle amount required by a task in a trace
WCEC	worst-case execution cycles

1. INTRODUCTION

The development of systems such as portable devices and Internet of Things requires low-energy design. For many embedded systems, low energy consumption is vital, and it has become a crucial feature in modern system design. Long battery life is required but devices are limited by battery size and weight. Heat dissipation caused by high power consumption is also undesirable and even unacceptable for wearable and implantable devices [1].

The application, supply voltage, execution frequency and process technology are the main factors affecting a device's power consumption [2]. Software and hardware can be optimized for low energy consumption and energy efficiency (Figure 1). Memory requirements and memory utilization optimization lead to lower energy consumption in software. Energy efficient memory technology and hardware design techniques reduce hardware energy consumption. Many techniques are related to hardware design. Design tools and methodologies are widely available. However, the impact of software should not be neglected.

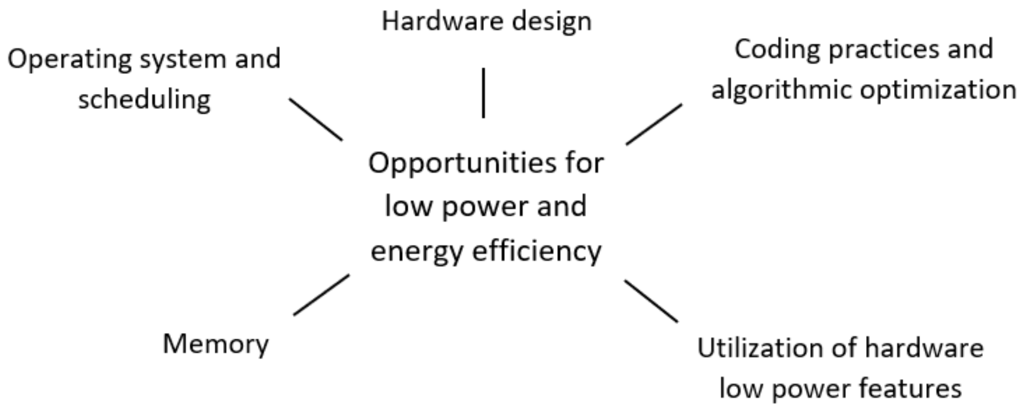

Hardware design
Operating system and scheduling
Coding practices and algorithmic optimization
Opportunities for low power and energy efficiency
Memory
Utilization of hardware low power features

Figure 1. Possibilities for low energy consumption in software and hardware.

The hardware design process consists of several levels of abstraction: technology, circuit/logic level and architecture level. Software decisions take place at higher levels of the design. Algorithm and system levels include software optimization and software-controlled power management. Techniques used to achieve low energy consumption and increase energy efficiency can be applied at different design abstraction levels from system to technology level. The greatest impact on energy consumption can be achieved with design decisions made at highest levels of abstraction, whereas the accuracy for power estimation decreases at higher levels [3].

Power management can be implemented in application code, by OS (Operating System) or with hardware (Figure 2). The existing hardware low-power features can be utilized, thereby using automated hardware control for power management. The advantage of software-based implementation is that it is configured for the application. RTOSes (Real-Time Operating System) used in today's embedded systems provide power management features and scheduling can be implemented for low energy consumption. [4]

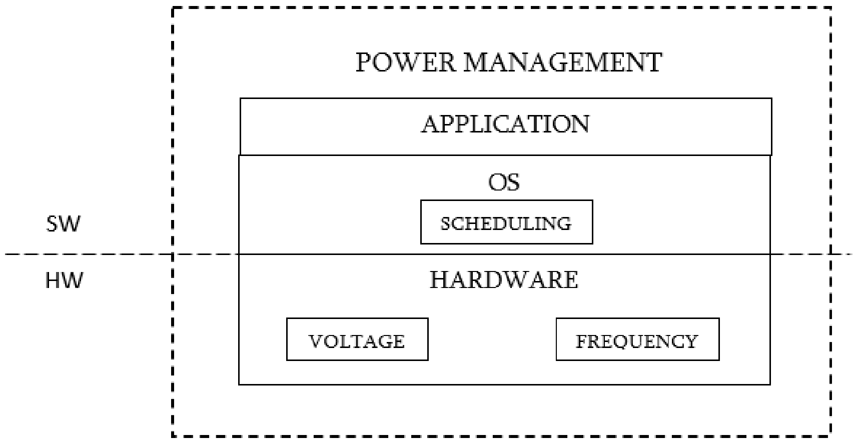


Figure 2. Overview of levels for power management.

Usually hardware and software are not designed in parallel and the design is done in separate teams rather than in an integrated team. In the context of this thesis, the hardware has already been designed and is capable of ultra-wide DVFS. The application software is provided by an external supplier. Thus, the method presented in this thesis for frequency optimization is the only remaining optimization technique which can be applied.

In Chapter 2 the design challenges for low energy consumption are discussed. Chapter 3 presents techniques used in software and hardware. It is necessary to understand the possibilities that the hardware and software provide and how these techniques affect the system behavior.

DVFS, a power management technique, is used to reduce the dynamic power consumption. Basically, clock frequency is dynamically adjusted while allowing the reduction in supply voltage. The reduction in dynamic power consumption comes with the cost of performance. However, a processor can spend much of its time waiting for events to occur [5]. The unused processing time is utilized by DVFS.

The effect of voltage and frequency scaling to task execution time and energy consumption is visualized in Figure 3. Instead of race-to-idle, task execution is prolonged. The challenge is that while low energy consumption is achieved, adequate performance level and correct behavior must be ensured. Execution time is especially critical in hard real-time systems where missing a deadline has fatal consequences. Thus, execution frequencies must be carefully chosen for an application utilizing DVFS.

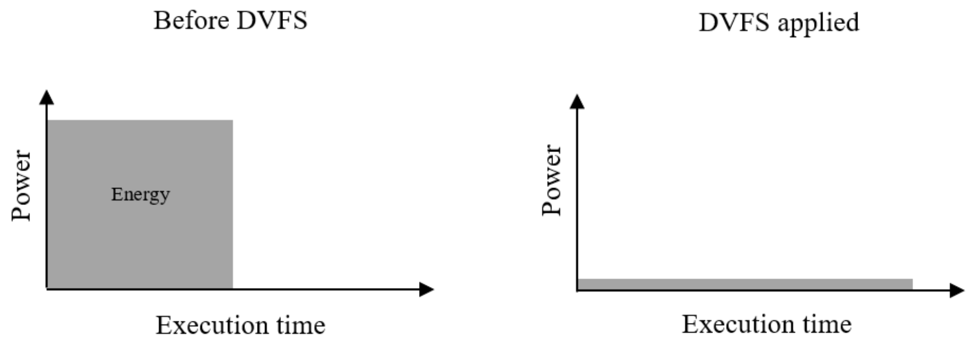


Figure 3. Basic idea of DVFS.

The objective of this thesis is to develop a method which can be used to define optimal execution frequencies for DVFS based on application analysis and profiling, leading to lower energy consumption while maintaining correct application execution. The method is developed for embedded real-time systems. The target system provides wide range of frequencies and is capable of near-threshold operation. Optimal execution frequencies are defined for inter-task DVFS algorithm. Thus, the frequencies must be defined at task-level. The method is developed based on the assumption that the execution frequency should be scaled as low as possible given the constraints.

In the context of this thesis, **software profiling and frequency optimization** are developed independently of the operating system. Many operating systems allow specific execution events to be traced. The requirement for the implementation is that tasks are executed as separate threads instead of a single thread. Only minimal code instrumentation should be required. The execution frequencies for tasks are defined based on trace data collected during execution and user-defined constraints. Software execution trace is used in profiling where data is extracted and used in frequency optimization.

2. DEVELOPMENT ISSUES

System energy consumption depends on several factors. The design challenges introduced by low energy design include degraded system reliability and timing overhead caused by power management. These issues need to be acknowledged as they cause incorrect system behavior or even failure. Energy and power benchmarking tools are also required.

2.1. Energy consumption

Power defines the rate at which energy is consumed. Average energy consumption E_{avg} is the time t integral of power P:

$$E_{avg} = \int P(t)\mathrm{d}t. \tag{1}$$

As a capacitor is charged energy is drawn from the power supply. Total energy drawn from the power supply when a capacitor is charged is defined using Equation (1) where the integral interval is the charging interval from 0 to timepoint T:

$$E = \int_0^T P(t)\,\mathrm{d}t. \tag{2}$$

Instantaneous supplied power is the product of supply voltage V_{dd} and capacitive current I_s. The charging interval is the interval when a capacitor is charged from 0 to V_{dd}. Thus, Equation (2) for energy drawn at each cycle simplifies to:

$$E = V_{dd}\int_0^T I_s(t)\mathrm{d}t = V_{dd}\int_0^{Vdd} C_L \mathrm{d}\,V_{out} = C_L V_{dd}^2, \tag{3}$$

where C_L is the load capacitance and V_{out} is the output voltage. Equation (3) shows the relation between supply voltage and energy consumption. Total energy consumption is affected by the rate at which charging is done. [6]

2.2. Timing challenges

As the correct behavior of an application depends on correct timing of the tasks during execution, overhead caused by power reduction implementation can severely affect the application behavior. Overhead is the time that the application execution is halted. The instructions related to power management are executed resulting in the necessary hardware configurations.

The amount of overhead depends on several factors: the hardware implementation characteristics and complexity of the power management algorithm. No matter what kind of approach is chosen for power management, each state transition always introduces some overhead. Depending on the state transition, the amount of overhead that reduces application execution time or increases response time can become significant. This needs to be acknowledged in application design as many systems require quick response to certain events which cannot be predicted.

Dynamic power reduction techniques can be divided into the use of power modes and DVFS. The two approaches have the opposite effect to the software execution but achieve reduction in power consumption compared to a state where no power reduction technique is applied (Figure 4). In that case, power consumption stays at the same level with task execution and idle. The system remains responsive.

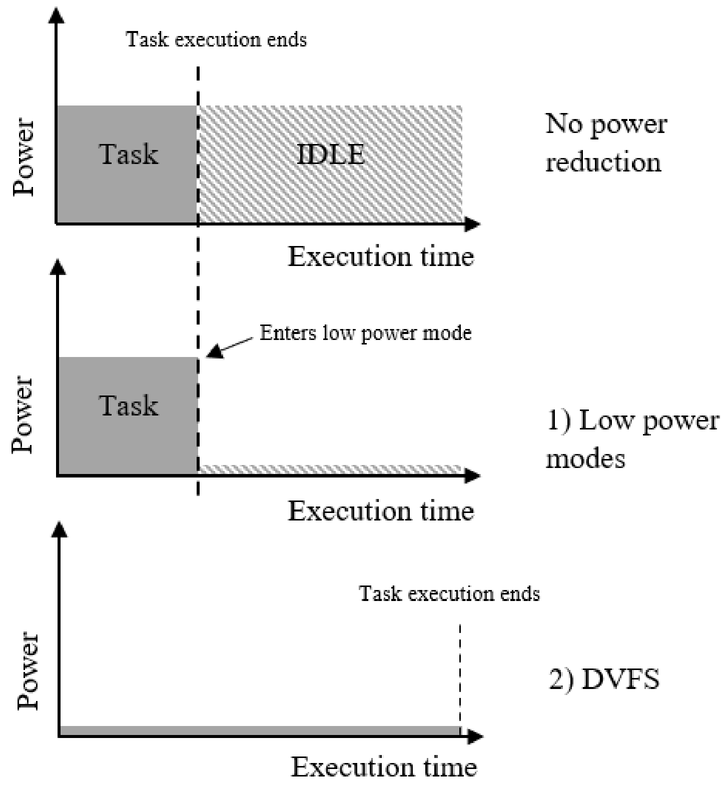


Figure 4. Changes in execution time and power consumption with no power reduction versus power modes (1) and DVFS (2).

2.2.1. Power modes

Race-to-idle approach is used with power modes. That is, the task being executed should be executed as fast and quickly as possible so that the processor can be set to a low power mode for the longest possible time period. These time periods where processor is not utilized are used to set the processor to a low power mode.

Low power modes vary from the active mode to low power or sleep mode to nearly being totally turned off or, as can be said, hibernated (Figure 5). The functionality is decreased as lower power mode is used. In active mode the processor executes at full speed and is fully powered. When moving to lower power modes, most clock signals are stopped, eventually stopping all clock signals. Some designs allow shutting down parts of the chip. The downside is the delay with every exit from and entry to a low power mode. Deepness of the power mode affects the wakeup delay and can cause unresponsiveness. [7, 8]

The time to enter and exit a light power mode, for example when waiting for an event, is very short [9]. A light mode does not lead to a great decrease in power. Deeper modes yield more decrease but increase the response time of the system. Some modes even require booting the processor [10]. It must be ensured that unnecessary wakeups are minimized, and the system remains responsive. The decision of when to enter a lower power mode must be chosen carefully. Thus, tradeoff between acceptable response time and achieved savings must be considered.

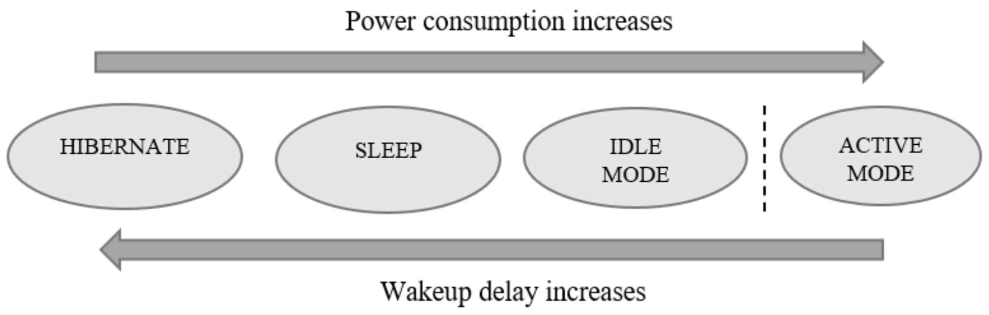


Figure 5. Power modes and their relation to power consumption and wakeup delay.

2.2.2. DVFS

DVFS tries to minimize the time spent in idle. Opposite to race-to-idle approach, energy consumed during active state can be decreased. DVFS controls execution frequency and voltage according to application requirements. The operation of the processor might not be reliable during these transitions, and the application software execution is halted [11]. To be able to estimate the amount of overhead that is introduced, the direction of the change must be known. The direction of the change corresponds to the value of the voltage and frequency levels used. The overhead depends on the direction that the OP (operating point) is changed as the order specifies whether execution frequency or supply voltage must be set first (Figure 6).

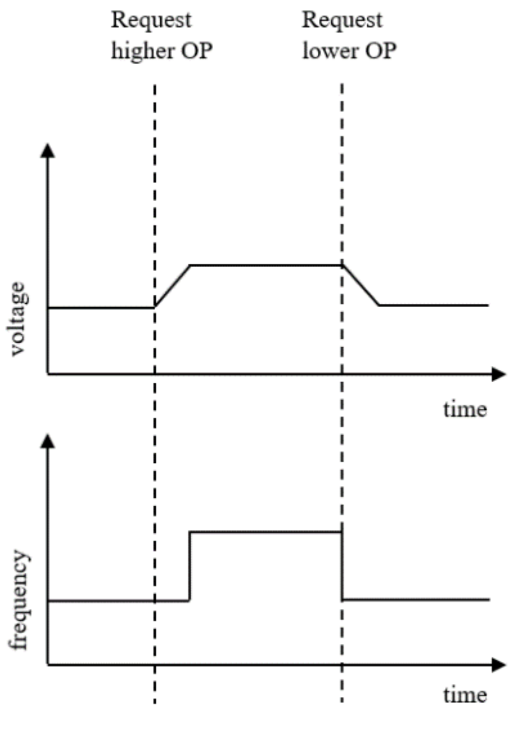


Figure 6. The order and delay of frequency and voltage change.

When frequency and voltage are changed to a higher level, the voltage must be set first to the requested value. Once the voltage is set to the value which can support the requested new frequency, the execution frequency can be set to the requested frequency value. On the other hand, when the change direction is the opposite i.e. voltage and frequency are changed to a lower level, frequency is set first. Then the voltage level can be decreased as the higher voltage level is no longer required. [12]

The software execution can be continued before the lower voltage level is reached when the change is done from a higher to a lower OP level. This is because the lower execution frequency is supported by the higher voltage level. When the execution frequency value is altered, frequency must be locked before the application execution can be continued. The same goes for voltage which must be set and at adequate level before execution frequency can be increased. The time taken to lock the correct frequency can be rather short. The setting time of the correct voltage is slower and defined by the slew rate [13]. Thus, the difference between the voltage levels defines some of the overhead introduced.

From application execution point-of-view, the overhead itself is not the only issue which results from frequency and voltage change. The change requires time and consumes energy. Thus, it is not feasible to be constantly changing the OP. The power reduction is achieved with decreased performance. How is the system performance affected if quick response is required but a low OP is in use? This is the case if an interrupt occurs. If the service to be executed is short, e.g. interrupt service routine, it is not feasible to change the OP as the execution time of the interrupt service routine should be short and could be shorter than the overhead time to change the OP.

2.3. Reliability

Low power design and harsh use environment conditions cause reliability issues. System becomes susceptible to functional errors and failure. Reliability defines the ability of a system to continue operation correctly even if errors occurs.

Low operating voltage and transistor scaling introduce timing skews. Timing errors arise from large process variability which increases as voltage is decreased, operation time increase due to aging, or transient errors caused by effects such as noise, power glitches and radiation. These errors are not permanent. They cause error in data or control flow in the software [14]. Data flow errors corrupt data values and cause output values to become incorrect whereas control flow errors cause incorrect execution order for instructions [15]. Time-independent reliability issues become more significant at lower voltages. [16, 17]

Redundancy is an approach used to make systems more fault tolerant. Several techniques for hardware and software redundancy have been developed. Redundancy techniques can be classified in structural, time and information redundancy. While hardware-based solutions are available, they always require additional hardware. Software-based methods are used to reduce the overall cost. Software-based methods introduce timing overhead, added code size and additional memory requirement. [18]

2.4. Benchmarking

Benchmarking tools are designed to measure the performance characteristics of a processor in a comparable manner. As low energy consumption can be achieved by compromising the performance, there is a need for power and energy evaluation. Tradeoff between power and performance can be assessed and balanced. Benchmarking eases the development process, but the application also has a great impact on the energy consumption of the system.

The Embedded Microprocessor Benchmark Consortium provides a benchmark for energy that targets ultra-low power microcontroller units called ULPMark. The benchmark quantifies optimization tradeoffs made between different energy demands and performance. Behavioral profiles are used to assess the device operation. ULPMark line consists of three different profiles. ULPMark-CoreProfile focuses on the energy of sleep and transitions between active and sleep mode. ULPMark-CoreMark is used to analyze performance and energy efficiency. ULPMark-PeripheralProfile can be used to assess the cost of most used peripherals. [19]

3. TECHNIQUES FOR LOW ENERGY CONSUMPTION

Several approaches are used to achieve lower energy consumption. These can be categorized into hardware techniques, algorithmic techniques, scheduling algorithms, and DVFS algorithms. Techniques from different categories can be combined for total system optimization for low energy.

3.1. Hardware techniques

Several design techniques which are applied at different design abstraction layers are used to lower energy consumption but they also introduce challenges (Table 1). Voltage scaling, frequency scaling, power gating and clock gating are the most commonly used techniques [20].

Table 1. Examples of low power techniques and their challenges

Abstraction level	Technique	Challenges
Architectural	Dynamic voltage scaling	Added overhead and increased delay
	Pipelining	Control overhead
Circuit / logic	Transistor sizing	Increased delay
	Clock gating	Increased logic
	Power gating	Increased delay
Technology	Scaling	Parameter variability
	Supply voltage reduction	Increased delay
	Threshold voltage reduction	Leakage power increase

Scaling of physical dimensions causes variability in device parameters such as threshold voltage due to process variation [21]. Supply voltage can be decreased near threshold or below the threshold voltage. However, current is not strictly cut off at the threshold voltage. The significance of static power consumption on system total power consumption grows. Susceptibility for soft errors is increased as capacitances and voltages are decreased. [22-24]

Another efficient technique to reduce dynamic power is clock gating due to clock tree constituting a large part of total power consumption. Clock gating prevents unnecessary charging by gating off clock supply from unused logic. Implementation requires additional control logic (Figure 7). Power gating has a similar approach as current is cut off to reduce leakage power. Although leakage power is efficiently reduced, it introduces delay and requires additional logic. [25-27]

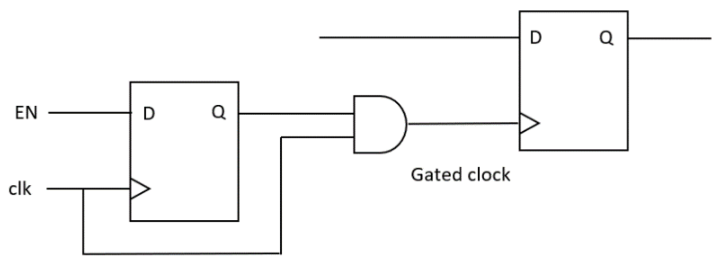


Figure 7. Basic clock gating cell.

Transistor width can be reduced to decrease dynamic power consumption. It causes increase in transistor delay and scaling decision is made based on tolerable delay. Switching activity is decreased by reordering transistors so that transistors which are likely to switch frequently are placed close to the circuit's output. [25]

Dynamic voltage scaling is applied at architecture level to reduce dynamic power. Reduction in supply voltage increases propagation delay. Glitches occur due signals in different paths not having the same delay [28]. Control can be implemented in hardware or software and the technique can be combined with dynamic frequency scaling.

Pipelining increases performance as instruction execution is divided into stages executed in parallel. However, conditional execution decreases performance and causes delay. Stalling can be avoided by using branch prediction. For wrong predictions, pipeline flushing is required wasting execution cycles before continuing with the right instructions. Pipeline structure affects the cost of wrong prediction and requirements for expensive registers. Another technique used at architectural level is low-power memory hierarchy. Memory must be optimized for the application specifically to reduce hardware usage. [25, 29]

3.2. Algorithm level techniques

Software optimizations are done for power consumption, execution speed and code size. Code efficiency decreases execution time. Data optimization techniques concentrate on minimizing the cost resulting from data transmissions and storage. Code can be optimized in multitude of ways, usually by a compiler, with example methods shown in Table 2.

Table 2. Common algorithmic optimization techniques

Optimization technique	Applied to	Description
Software pipelining	Loops	Reduce execution time by changing loop execution order and utilizing instruction level parallelism.
Precision reduction	Arithmetic operations	Reduce computational precision and decrease energy.
Assembly and intrinsic functions	Functions	Library function calls are replaced with low-level instructions decreasing execution time.
Strength reduction	Expensive operations	Expensive operations are replaced by less expensive ones.

The algorithm implementation can be evaluated by examining the instructions that are needed for the operations. Instructions have different effects on the power consumption as they activate different functional units. The aim is to minimize the most frequently executed paths to reduce hardware switching activity. Functions can be transformed by using different commands to perform the same functionality. Data patterns and code sequences which consume less power are key for low-energy optimization. [20]

First, the best suitable algorithm for the given problem must be chosen. Same functionality can be achieved with variety of algorithms. The best algorithm can be multiple times better than the worst algorithm, whereas implementation details result only in improvement of a few percent [30]. The algorithm is chosen so that it is efficient for the available hardware. It includes computations and storage. An algorithm transformations and implementation should map to the computational resources available.

Basic programming techniques which can increase code performance include the use of fixed-point arithmetic when there is no support for floating point operations. Integer values should be correctly sized for the processor usually according to the natural word length. For data structures, the parameters should be aligned accordingly. [30]

Instruction selection and ordering are optimization methods. Instruction selection consists of selecting the lowest power instruction sequences. Instruction scheduling is used minimizing energy consumed as processor switches execution from one type of instruction to another i.e. the circuit state without affecting the program behavior. These are followed by register allocation which is discussed in memory-based techniques. Switching activity caused by an operation can be tried to be minimized by operand swapping. [31]

Stand-alone assembly, inlining and intrinsic functions are used to replace high-level language code. Assembly code lacks portability, can inhibit compiler optimizations and be a laborious task to implement. Intrinsic functions are an alternative for developed-implemented assembly. They are used as standard library functions. Intrinsic functions appear in the code as standard function calls but are replaced by the compiler with a low-level implementation. Intrinsic functions which are compiler specific are built into the compiler. Processor vendors also provide platform-specific intrinsic functions. These optimizations eliminate function call overhead and reduce execution cycles. [32]

Function call overhead can also be decreased by eliminating recursion in procedure calls. Recursion can be replaced with a loop structure. For each function call, general context of the function such as variable values must be pushed onto the stack. Another technique to reduce function call overhead is to use inlining which replaces the function call with the function body.

Loops provide opportunity for code optimization. Loop unrolling is used to optimize performance and power, but tradeoff is increasing code size. In practice, loop unrolling means that a loop is partially unraveled by reducing the number of control statements. Loop unrolling increases code parallelization and efficiency. It is beneficial with long pipelines and loops with a lot of iterations. The total unraveling is not beneficial as it leads to large code size and increased memory accesses.

Another technique for loops, which can also be combined with loop unrolling, is software pipelining which reforms loop operations to decrease execution time. Software pipelining uses instruction level parallelism. Loop iterations are reordered for out-of-order execution. Software-pipelined iterations are formed of instructions from different iterations of the original loop. Iterations are reordered so that dependencies are removed. Instructions from different loop iterations can be executed in the same pipeline.

Precision reduction is a technique which can reduce cycles required to execute a function and is applied to mathematical functions by reducing precision without degrading the output value for the given task. High precision leads to the used of more functional units and requires more cycles for execution. This technique can be applied if precision is too high for the purpose. [20]

Replacing integer division with a constant multiplication is an example of strength reduction. Less expensive but similar operations are used to replace the expensive ones. Common subexpression elimination is another compiler technique which eliminates redundant calculation by identifying expressions which result in the same value and storing them into another variable which can be used to reformulate the code. [33]

3.3. Scheduling

Scheduling is one core task of an operating system. Although scheduling can be implemented even on bare metal, operating systems provide various algorithms for scheduling and ready framework for implementation. Scheduler decides which task is selected for execution at a scheduling point based on a scheduling algorithm (Figure 8).

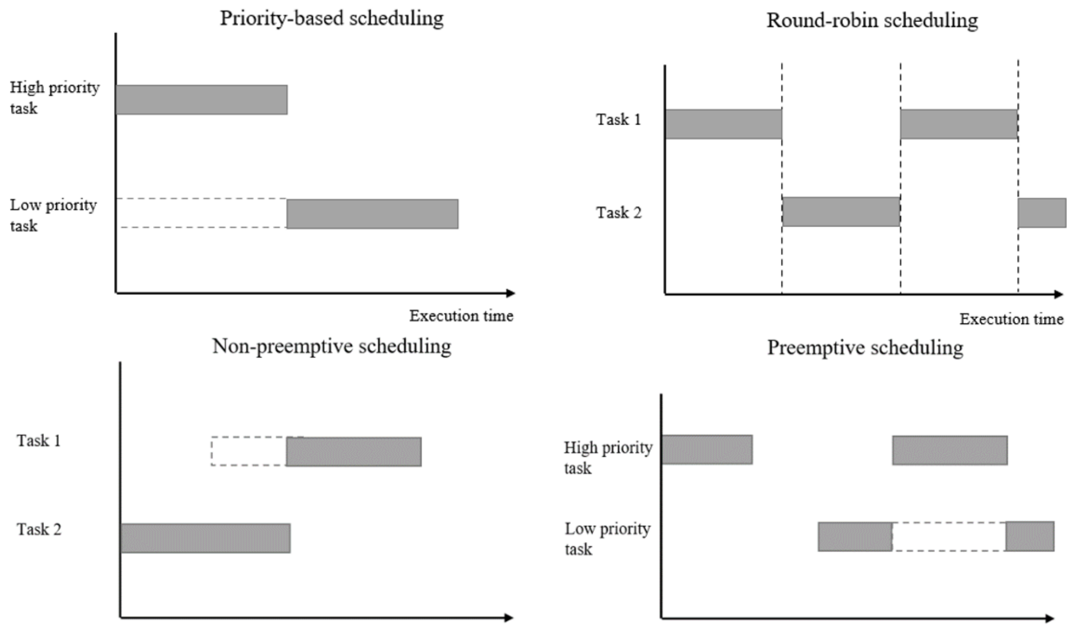


Figure 8. Execution time allocation in priority-based, round-robin, non-preemptive and priority-based preemptive scheduling.

Scheduling policies can be divided into priority-based scheduling and round robin scheduling. In priority-based scheduling the task with the highest priority is given execution time first whereas in round-robin scheduling tasks are executed in turns. Tasks can also be preemptive. In priority-based scheduling, the execution of a task can be interrupted by another task with a higher priority i.e. a context switch occurs before the task has finished all its jobs. This is preferred in real-time systems as non-preemptive scheduling lacks the ability to react to real-time events.

EDF (Earliest deadline first) and RM (Rate monotonic scheduling) are optimal algorithms. EDF is an optimal scheduling algorithm for a single processor: if a task set can be scheduled, it can be scheduled with EDF. The task with the shortest time until its deadline is selected for execution. With dynamic priority assignment, the priorities are assigned online. Processor can be fully utilized i.e. the utilization bound is 100 % for EDF. [34, 35]

Fixed priority scheduling is the most implemented approach in commercial RTOSes [36]. Task priorities are defined by the developer at compile-time, usually according to task timing requirements, and remain the same during execution. RM is a fixed priority-based scheduling algorithm. A task set cannot be scheduled with any fixed-priority algorithm if it cannot be scheduled with RM. Task priorities are assigned indirectly proportional to task periods. Tasks which have short periods and execute more frequently are assigned higher priorities. For RM, the utilization can drop even to 70 %. The problem for these utilizations bound definitions is that task sets are assumed to consist of only periodic and independent tasks. [35, 37]

In practice, many tasks have inter-dependencies. Many task sets contain aperiodic and sporadic tasks, and it can be hard to predict the arrival times of these tasks. Also, in systems like Internet of Things, telecommunication protocols set timing constraints which have to be met. Energy harvesting real-time systems pose another scheduling problem as the challenge is to operate so that energy can be harvested as much as it is consumed. The available energy and processor utilization need to be considered in power management implementation. [38]

The key for power-aware scheduling is providing just enough performance and power for the workload of a processor any given time while maintaining deadlines. Any slack time arises from suboptimal scheduling as 100 % utilization is not achieved. Power-aware scheduling can utilize knowledge of energy models related to tasks and their resource utilization.

3.4. DVFS algorithms

The target of DVFS is to keep the processor utilized. Although the idea of applying DVFS is very simple, it can be a complex task to apply in practice. As real-time systems have timing constraints, frequency cannot always be scaled to the lowest level. These timing constraints are used to define the execution frequencies.

DVFS can be classified different ways (Figure 9). They can be classified by their implementation, i.e. the time the OP value decisions are made, and whether frequency is changed only at task borders or within tasks. Inter-task algorithms are coarse grained as the application is only evaluated at task-level usually relying on knowledge of task worst-case execution. They introduce less overhead and less code instrumentation. Intra-task algorithms switch OP within task boundaries. This enables more fine-grained approach to tuning the execution frequency. Also, interval-based online approaches exist [25]. [39]

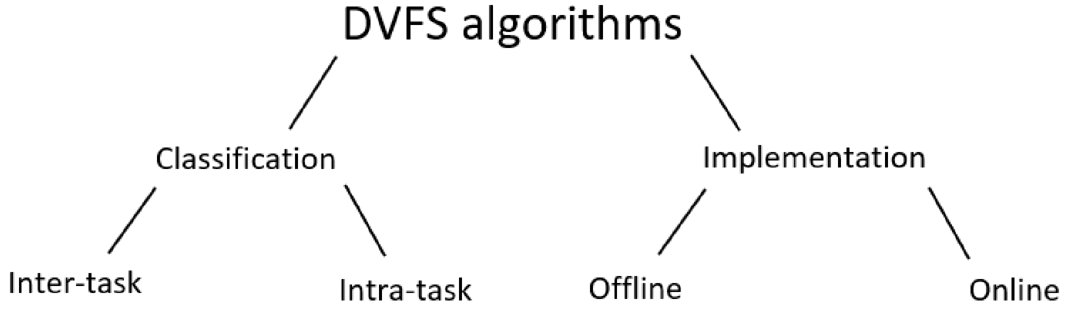


Figure 9. Classification of DVFS algorithms.

Slack is the time by which an execution of a task can be delayed without violating its timing constraint. Two types of slack can be utilized in frequency scaling (Figure 10). Dynamic slack arises when the actual execution time of a task is shorter than the worst-case execution. Offline algorithms define the values at compile time. These techniques are highly predictable as there is no adaptation to runtime variation. Online algorithms make frequency decisions at runtime and make use of runtime variation in task execution. [39]

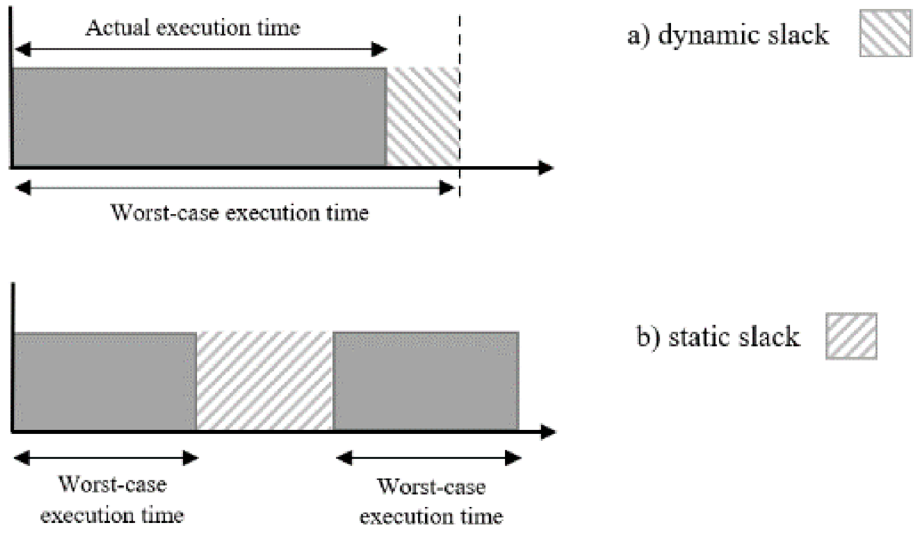

Actual execution time
Worst-case execution time
a) dynamic slack
b) static slack
Worst-case execution time
Worst-case execution time

Figure 10. (a) Dynamic slack arises when a task requires less than worst-case cycles. (b) Static slack can be present even if tasks require their worst-case cycles.

The downsides of online algorithms are increased complexity and timing overhead. Decisions are made based on past execution or predictions of future execution and it is hard to ensure deadlines. The decrease in power consumption can be lower with offline algorithms. There is no adaptation to varying execution and only static slack is utilized. The actual execution cycles required for a task completion can't be known until the task has finished execution. Thus, deadlines are guaranteed by assuming worst-case execution.

For DVFS to be efficient, the energy overhead and timing overhead should be minimal. The amount of OP switches should be kept minimal as it adds delay to application execution. The decrease in energy consumption depends greatly on the target system used. The amount of operating frequency levels that are available directly affects DVFs efficiency. The ability to use low voltage which are set according to operating frequency improves efficiency. Processor utilization can be used as a measure for DVFS performance. Thus, scheduling has a great impact on the performance.

Several algorithms have been proposed in literature. Only algorithms for a single core are discussed here. They assume either discrete or continuous frequencies. Depending on the algorithm, tasks can be assumed to be periodical, aperiodic or sporadic. The frequency decisions can be made based on static code analysis and data collected during runtime. Most of the algorithms discussed are developed jointly with EDF or RM scheduling algorithm. Table 3 gives overview of different uniprocessor algorithms for real-time systems.

Table 3. Comparison of DVFS algorithms proposed in literature for real-time systems

Algorithm / Developer	Classification	Scheduling	Frequency	Frequency decision	Challenges
Average rate heuristic	Online	EDF	Continuous	Defined for an interval at a time. Required total amount of execution cycles in the interval defines frequency value	Deadlines misses may occur when additional tasks are released for execution within an interval
Static	Offline	EDF	Discrete	One execution frequency for the whole task set with timing constraints	Single frequency suboptimal for DVFS
Cycle conserving	Online	RM / EDF	Discrete	Current utilization and assumption of future worst-case requirements updated at each scheduling point	Ensuring deadlines when frequency is decreased
Look-ahead	Online	EDF	Discrete	The lowest frequency for current task to meet a deadline	Ensuring deadlines as low frequency used for execution
Bini et al.	Offline	EDF / Fixed priority	Discrete	Single, lowest frequency for the task set under given scheduling algorithm which makes schedule feasible	No adaptation to runtime variation, single optimal frequency defined
Pinheiro et al.	Online	-	Discrete	Initial frequencies for each task defined offline and lowered within a task if worst-case path not taken	Increased code instrumentation
Energy-aware DVFS	Online	EDF	Discrete	Available energy: stored and harvested in the future.	Energy overhead is neglected for frequency/voltage change.
Hybrid DVFS with reinforcement learning	Online	-	Continuous	Several DVFS algorithms used. The most suitable algorithm for given system state is learned.	Requires training for optimal adaptation. Online algorithm adds increased overhead.

One of the first papers proposing frequency scaling algorithms for power-aware scheduling introduced algorithm Average Rate heuristic. It is developed for aperiodic tasks with a known deadline and an arrival time. Based on execution cycles required for tasks executing within a scheduling interval, an average-rate requirement is defined. Based on the average-rate requirement, the execution frequency is defined for the given time interval. EDF is used to schedule the tasks. [39, 40]

Pillai et al. integrated three DVFS algorithms to EDF and RM schedulers. Here, it is assumed that deadlines equal task priorities so that Liu and Layland utilization bound can be used to check schedule feasibility [41]. Tasks are assumed to be independent. Static voltage scaling is a simple algorithm which only defines one execution frequency for the whole task set which maintains schedulability. Only static slack is considered. Cycle conserving algorithm was created based on the assumption that tasks usually require much less execution cycles for completion than worst-case. This slack is utilized by other tasks. At first, it is assumed that a task requires its worst-case execution cycles and high execution frequency is used. Early completion enables to lower the execution frequency for the remaining scheduled tasks. Frequency can also be increased to meet the timing constraints. Look ahead algorithm is proactive, and execution is started with low frequency according to minimum amount of cycles required for all tasks. This enables to execute at low frequency and only increase execution frequency if required for meeting deadlines. [42]

A framework for DVFS implementation with EDF and fixed priority scheduling was proposed by Bini et al. Continuous frequency range was assumed to define the frequencies offline. A speed modulation technique was proposed which is used to achieve the desired frequency using two discrete frequency values. Only single optimal frequency is defined for the task set which makes the schedule feasible under the chosen scheduling algorithm. In addition to considering mixed task sets, scaling overhead and energy overhead are also considered in the model. [41]

Pinheiro et al. developed a methodology to turn a regular source code into power-aware code by inserting DVFS control code into it. Static source code analysis is applied to study the execution paths inside a task which can be used to create a control flow graph and define the worst-case execution path. This knowledge is used to ensure schedulability and define initial frequencies for tasks. Execution frequency within a task can then be reduced when worst-case path is not taken. Task response times are used as new deadlines. The method also considers task interference and overhead. [43]

Energy-aware dynamic voltage and frequency selection algorithm is proposed for energy harvesting systems. The operating frequency is selected based on the available energy and the energy that will be harvested soon. The frequency is decreased if there is not enough energy. Otherwise, tasks are executed with the highest frequency. Task execution is delayed until enough energy has been harvested for task execution which may lead to deadline miss. EDF is used for scheduling and with infinite amount of energy, the energy-aware algorithm reduces to EDF. [44]

Hybrid DVFS scheduling with reinforcement learning is more recently proposed algorithm for real-time systems with machine learning based control algorithm. Adaptability to execution variation is achieved with Q-learning where the most suitable action to be taken is chosen for given system state. The core idea is that a single algorithm is not optimal in different execution scenarios. A new algorithm is not proposed but already existing DVFS techniques are used. The system state is observed, and the most suitable algorithm is chosen during execution at every scheduling point. System state is described with system utilization and dynamic slack for the given task set. Q-table is defined for each system state and algorithm, containing penalties for each action. Penalty functions is defined based on average energy consumption in a hyperperiod. [45]

Temperature-aware DVFS algorithms have been proposed to reduce thermal problems caused by heating. As a result of thermal management, reliability increases. Temperature-aware DVFS algorithms for real-time systems consider both timing and thermal constraints. Approach can be to optimize power or performance. [46]

3.5. The role of software

Software has the most significant role in the optimization of a system's energy consumption. It has been estimated that software accounts for approximately 80 % of a system's energy consumption [47]. This is due to hardware activity which is controlled by software. Software provides great opportunities to gain even more energy savings when designed specifically for energy efficiency. On the contrary, inefficient use of the hardware causes waste of energy i.e. inefficient software wastes the advantages of hardware designed for low energy consumption [48]. The problem is not solved by designing more energy-efficient hardware. Thus, software and hardware both need to be specifically designed for low energy consumption.

The use of an operating system makes application development and integration less complex. Today, RTOS is used in almost two thirds of embedded systems [49]. In contrary to GPOSes (general-purpose operating systems), many embedded systems use an RTOS due to their real-time requirements which can't be guaranteed with GPOSes.

Many well-known operating systems, although not classified as RTOS, provide power management features. For example, Linux provides framework for DVFS called CPUFreq and the utilization of low-power idle states with CPUIdle [50]. For examples of RTOSes, VxWorks, FreeRTOS and Mentror Graphics Nucleus RTOS provide at least some form of feature for low power such as tickles idle or power management frameworks.

There are different approaches how energy-efficient software implementation could be done. The most suitable option depends on the application and more than one technique can be used for the same application. Static techniques implemented at compile time or good programming practices are not enough for total energy optimization but can be used to optimize the code for efficiency. As the software is executed on the processor, it is the only component that can be used to achieve real energy savings. By utilizing software to manage CPU (central processing unit) features that affect the power consumption of a system, lower energy consumption can be achieved [51].

Techniques for writing efficient software which increases performance and decrease memory requirement are well known among software developers [52]. However, as energy consumption has become a critical design factor, there is a need for guidelines on how to design software for energy consumption minimization and tools which can accurately model the energy consumption of software design decisions.

4. PROFILING FOR THE NEAR-THRESHOLD PROCESSOR

A method for defining execution frequencies for a DVFS-enabled, near-threhold processor is presented. A software execution trace is used as a base for software analysis. Extracted data from the execution trace is used for profiling. The data is combined with user-defined data, such as task timing constraints, for frequency optimization. The software events which are traced are chosen so that they can be traced with any operating system. Requirement is that tasks are executed as separate threads. The goal is to minimize application code instrumentation so that minimal work is required to collect the execution trace.

The algorithm defines execution frequencies at task-level. Tasks in optimization will have single frequency values assigned. The object is to scale the execution frequency as low as possible assuming this leads to decrease in energy consumption. To generalize the problem, it is assumed that task execution times scale linearly with the execution frequency.

The frequency optimization model is mainly intended to be used with hard-real time systems whose task execution is predictable and tasks have deadlines. That is why execution times are defined based on execution cycles that are required to complete the task at worst-case scenario. The operating systems used in this work is FreeRTOS. Tracing is implemented with trace macros. The profiling and optimization are developed independently of the operating system. Optimization can be done without the execution trace if software execution properties are known.

The frequency optimization can be divided into three phases: software execution tracing, software profiling and optimization. Overview of the implementation is given in Figure 11. Implementation is parameterized so that profiling and optimization related parameters are not fixed for the given system and trace event implementation.

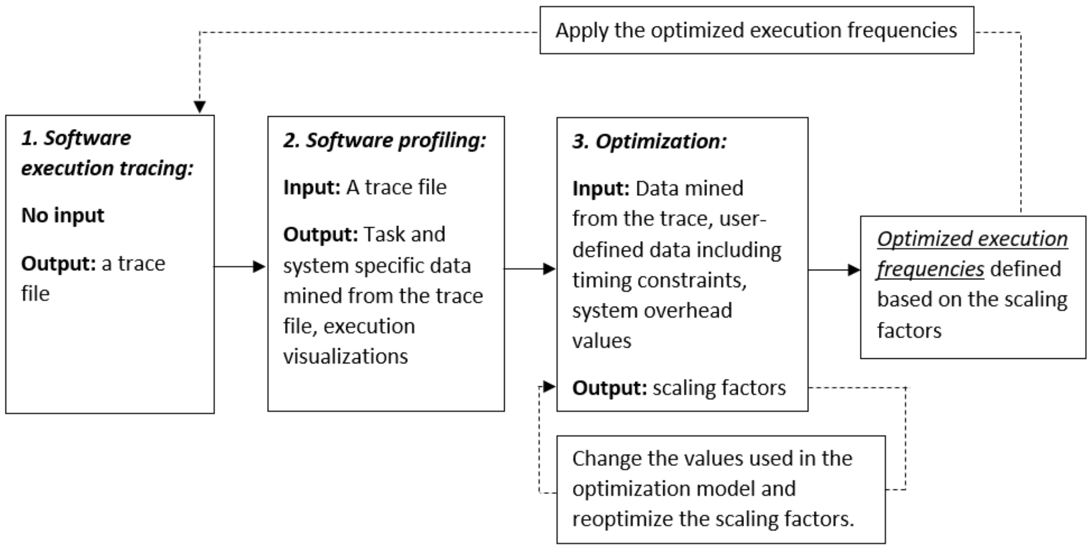

Apply the optimized execution frequencies
1. Software execution tracing:
No input
Output: a trace file
2. Software profiling:
Input: A trace file
Output: Task and system specific data mined from the trace file, execution visualizations
3. Optimization:
Input: Data mined from the trace, user-defined data including timing constraints, system overhead values
Output: scaling factors
Optimized execution frequencies defined based on the scaling factors
Change the values used in the optimization model and reoptimize the scaling factors.

Figure 11. Overview of the frequency optimization.

Software execution tracing includes the actual tracing and some preprocessing steps before execution trace is profiled (Appendix 1). Data required to form the optimization model is mined from the trace in profiling phase. Task execution and task execution properties can be visualized in the profiling phase to give insight of the application execution characteristics. Software profiling and user-defined data is used to form an optimization model which models task execution of the application. The model includes timing constraints and overhead of DVFS. The optimal execution frequencies for the application tasks are derived from the optimization results.

4.1. Task execution

The task set of an application is modeled with a set of parameters. These parameters reflect task execution properties and are used to create the task execution model. Each task in the set is characterized by execution time, deadline and task period or minimum inter-arrival time. The number of task releases is defined with a task instance value for each task. The parameters related to task execution are visualized in Figure 12. The periodicity of a task can affect the performance after optimization if task execution cannot be modeled adequately.

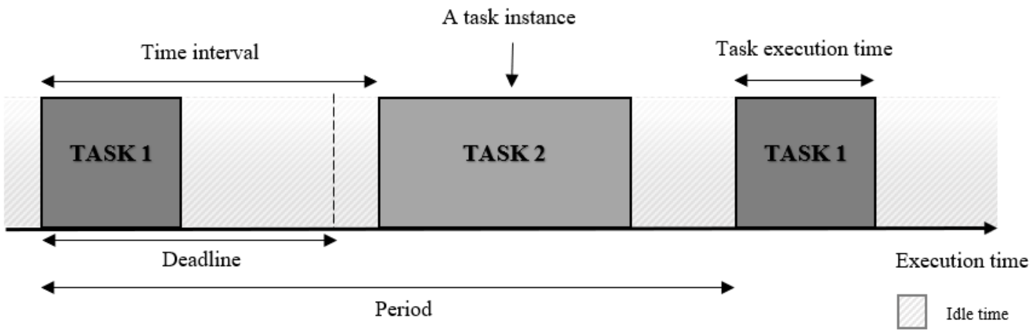


Figure 12. Task execution related parameters.

The execution time of a task is defined by its execution frequency and required execution cycles to complete. A time interval defines the amount of time from a task release to the next task's release. Tasks are assumed to be independent. This means that there are no interdependencies between the tasks e.g. a task must wait for another task to complete before it can be executed.

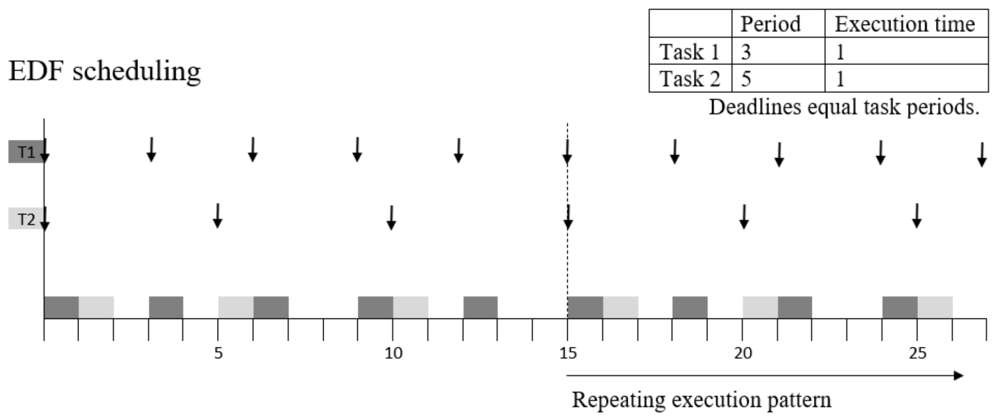
WCEC (worst-case execution cycles) is used to define the task execution time with the respective frequency. WCEC value defines the maximum number of execution cycles that a task requires to complete. It is not trivial to know if the worst-case has occurred in the trace and if the worst-case execution time is defined correctly. Knowledge of the application and task worst-case conditions are required from the user so that the worst-case conditions can be produced. On the other hand, the execution time for a task must be known, as an undersized value can lead to deadline misses.

This deadline value must be provided by the user. A deadline can be defined based on implementation requirements; within which time limit a certain task must be completed. If there is not known deadline, time interval value can be used instead of a task deadline value. In some cases, deadline can be defined based on subjective experience.

The reason why a context switch occurs is not visible in the trace. Each instance of a task is treated as the task has executed all its jobs i.e. no preemptions occur during the trace. Tasks are usually implemented as infinite loops. It is not straightforward to define the points where a task starts and when it has completed all its jobs.

Task execution can take different execution paths which lead to different operations and different execution cycles. It is assumed that after completing its jobs, the task will be left to wait for an event, and this is defined as one whole execution of a task. Task is switched out if conditions to continue the execution of the task are not met. When the event has occurred, the task is moved to ready state and when it is switched in, it can continue the execution. The idle task is an exception. It has the lowest priority and is always ready to execute as there should always be one task ready to be executed.

A hyperperiod can be defined for a periodic task set. Hyperperiod is the smallest time interval after which the task execution pattern is repeated. An example of a task set execution pattern is given in Figure 13. Ideally, a hyperperiod would be used as a trace section for optimization. A hyperperiod can be defined as the least common multiple of the task periods [53]. It might not be possible to define a proper hyperperiod if tasks are not periodic or the hyperperiod can be long.



	Period	Execution time
Task 1	3	1
Task 2	5	1

Deadlines equal task periods.

Figure 13. Example task set with a hyperperiod of 15.

4.2. DVFS implementation

The assumption that the execution time scales linearly with the execution frequency is made because task execution cycles are the only measure which is used to describe a single task instance's execution and it is not analyzed if a task is CPU-bound. The contents, execution paths and decomposition of a task are not included in the frequency optimization model. This also leads to the assumption that lower energy consumption is achieved at lower frequencies as only CPU power consumption is considered. The optimization results define new execution frequencies.

The OP is defined as a voltage-frequency pair. The frequency levels are discrete values and the frequencies that can be used depend on the system. For the implementation, it is known that a large range of frequencies can be generated and the step size between frequency levels depends on the point in the frequency range. The step size increases with higher frequencies. The frequencies are always rounded upwards if the exact frequency cannot be generated. This is because it must be guaranteed that the timing constraints are not violated because the frequency is falsely rounded. Due to these restrictions in the optimization model and DVFS implementation, the actual frequencies differ from the exact optimized frequency values.

The OP values for each task are defined at compile time. The number of different frequencies is not limited in the optimization; thus, the number of OPs is not limited. All the tasks do not have to be assigned with an OP. During frequency optimization, user can select the tasks which are included in the optimization. This means that only these tasks will have execution frequencies defined. It is left to the user to decide which tasks are included in the optimization. It is proposed that short tasks which execute frequently should be excluded due to overhead. Left out tasks will execute with the OP in use at given time.

4.3. The optimization model

The data collected from the execution trace in the profiling phase and user-defined data is used to form the optimization model. Linear optimization problems are convex problems. Thus, a linear optimization model is chosen because it is known that when a solution is found it is also the global optimal solution [54]. The optimization is implemented in MATLAB using linprog solver function. Description of optimization input values are provided in Table 17 and Table 18 in Appendix 1.

Scaling factors s_i are the decision variables in the optimization problem. Scaling factor values are defined based on the optimization constraints and the profiling parameter values. Only the tasks that are included in the optimization will have a scaling factor defined. Values are defined for each task individually. The number of scaling factors and optimized tasks is not restricted. The relation between scaling factor s, optimal execution frequency f_{scaled} and task execution frequency f is

$$f_{\text{scaled}} = \frac{f}{s}. \tag{4}$$

The optimization model is formed as a maximization problem. The objective function defines the total task runtime which is to be maximized. The target is to scale the frequency low within the timing constraints which should maximize total task runtime. The tasks that are included in the parameter definition are included in the optimization. The tasks for optimization are selected from a list of all tasks found in the trace if trace file is used as a base for optimization. Otherwise, the number of tasks must be defined before task parameter values can be given.

Runtime for a task instance is defined by the worst-case execution cycles *wcec* and the execution frequency *f*. Context switch overhead is included in the worst-case execution cycles. The total runtime of a task in the section is estimated by multiplying the runtime of the worst-case task instance with the number of task instances c_i in the given trace section. The total runtime which is to be maximized is the sum of all tasks' runtime estimations defined as

$$\text{Max. } \sum_{i=1}^{n} c_i \times \frac{wcec_i}{f_i/s_i}. \tag{5}$$

One constraint that limits frequency scaling is the total time available for task execution. Time amount t_{avail} is the difference between trace length and user-defined reserved time value for the whole time period. Overhead related to voltage and frequency change for each task instance also limits total task execution time. V_{lock} defines voltage settling time and F_{lock} defines frequency lock time. The constraint for total runtime is

$$\sum_{i=1}^{n} c_i \times \frac{wcec_i}{f_i/s_i} \leq t_{\text{avail}} - \sum_{i=1}^{n} c_i \times (F_{\text{lock}} + V_{\text{lock}}). \tag{6}$$

Each task is assigned a relative deadline dl_i. Ideally, execution frequency would be scaled until execution time corresponds to the deadline value. For each task, voltage and frequency change overhead and user-defined reserved time amount decrease the maximum execution time for a task. Guidelines for defining reserved time amounts are discussed in Subsection 4.5.3 User-defined data. These constraints define that task execution times is

$$\frac{wcec_i}{f_i/s_i} \leq dl_i - (F_{\text{lock}} + V_{\text{lock}} + t_i). \tag{7}$$

The decision variables, the scaling factors, are bounded by upper and lower limits. The limits for a scaling factor are defined separately for each task as

$$s_{i_{\max}} \geq s_i \geq s_{i_{\min}},\ i=1,\ldots,n. \tag{8}$$

The lower bound for a scaling factor defines the highest frequency that can be produced. As the chosen approach is to scale the frequency as low as possible, the lower bound is defined by default as one. If scaling factor value is one, the frequency of a task is not scaled.

The upper limit for a scaling factor can be defined based on the system used if there is no reason to limit the frequency for a task. A reason to limit the task execution frequency could be for example the use of an expensive I/O (input/output) device or timing requirement of a protocol. Expensive is used here to portrait the energy consumption of the system. If frequency scaling upper bound is only defined based on the system, the lowest frequency level that can be produced and current execution frequency for a task are used to define the upper bound for the scaling factor.

4.4. Software execution tracing

Software execution is traced to collect information on the software execution behavior. The collected data is used to analyze and model task execution characteristics. Three events related to task execution are traced. These events are successful task creation, task switch in and task switch out. FreeRTOS OS provides trace hook macros which are used to collect runtime data [55]. The trace hook macros are included in the FreeRTOS source code as empty macros.

A different trace macro is implemented for each of the events traced and each event is indicated with a specific abbreviation. All trace events contain the respective cycle count register value for the traced event. Passed time can be defined when the execution frequency is known. Additional information in the trace event print varies depending on the event and each information field is separated by a colon. Event print contents are described in Table 4. Quotes indicate that the text is replaced with a respective value in the trace.

Table 4. Trace event print descriptions

Event name	Event abbreviation	Trace print			
Task create	TC	TC	"Task name"	"Task handle"	"Cycle count"
Task switched in	CS-I	CS-I	"Task handle"	"Frequency"/CC	"Cycle count"
Task switched out	CS-O	CS-O	"Task handle"	CC	"Cycle count"

The trace macro for task creation is called when a task is successfully created. Trace event is implemented with traceTASK_CREATE macro. Once a task is created, a task control block containing data about the task is created. It is not necessary to include task create events in the trace for profiling although it is the only event where the descriptive name for a task is visible. Descriptive task name is used in profiling data and visualizations produced from the trace. Tasks will be named unknown if the names are not available and user must be able to identify the tasks during optimization.

The task-switch-in macro is called when a task is selected to enter the running state where the task is executed and utilizing the processor. This event defines the start for task execution. The task handle defines which task is being executed. Trace event is implemented with traceTASK_SWITCHED_IN macro. Pseudo code for the implemented task-switch-in trace macro is given in Figure 14.

```
void traceTASK_SWITCHED_IN() {
        dvfs_callback();
        frequency = getCPUFrequency();
        event_switch_in(frequency);
}

void traceTASK_SWITCHED_OUT() {
        dvfs_callback();
        event_switch_out();
}
```

Figure 14. Pseudo code for task-switch-in and task-switch-out macro definitions.

First function call in the trace macro is a DVFS callback function. This function is used to set the execution frequency and supply voltage i.e. the values for an OP. The execution frequency of a task is requested only if OPs are used. Frequency is requested after it has been set for the given task. Task handle and cycle count register value are read within the trace print function call. Due to this implementation, frequency and voltage settling cycles are excluded from task execution cycles.

The task-switch-out macro is called just before a task being executed is switched out. Trace event is implemented with traceTASK_SWITCHED_OUT macro (Figure 14). Switch-out macro includes a DVFS callback function before event print is taken. The cycles taken by the callback function are included in the task execution cycles as the cycles are read after the callback, in the trace event print function. OP used is not changed in switch-out trace macro. Thus, settling delays are excluded from task execution cycles. The handle in the event trace print contains the handle of the task that is just about to be switched out. This event defines the endpoint of task execution.

Profiling implementation in MATLAB requires the input trace file to have a specific form. The trace file is first processed with a Python script (Figure 26 in Appendix 1) to remove any unnecessary data. Knowledge of the trace event print structure is utilized in preprocessing. An example of a processed trace is given in Figure 27 in Appendix 1.

4.5. Profiling

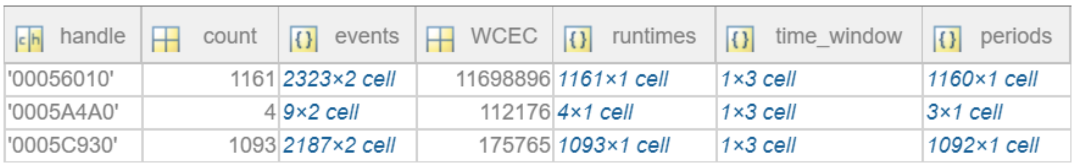
Software profiling is a form of dynamic program analysis. It is used to analyze data collected of software execution to examine software execution behavior. In profiling phase, the main object is to collect data which is useful for the optimization and is also used for execution trace and task parameter visualization. In this profiling implementation all variables presented in Table 5 must be defined to match the system overhead and implemented trace prints before profiling and optimization. Once the values are configured, trace file can be loaded for profiling.

Table 5. Variables which values must be defined before profiling and optimization according to used system and tracing implementation

Variable name	Description	Default value
Variables related to trace prints and MATLAB implementation		
switch_in	Value must match the abbreviation used for context switch in event.	CS-I
switch_out	Value must match the abbreviation for context switch out event.	CS-O
task_create	Value must match the abbreviation for task create event.	TC
fields	The number of data fields used for events. All event trace prints must have the same number of data fields.	4
CYC_COUNT	Abbreviation used in the 3rd data field if frequency information not included.	CC
idle_task	Name of the idle task in the used OS.	
Variables used in the optimization model		
volt_change	Time in milliseconds which is assumed to be required for voltage settling.	-
freq_change	Time in milliseconds which is assumed to be required for frequency locking.	-
COST_CS_IN	The total overhead in milliseconds which is sum of frequency locking and voltage settling.	volt_change + freq_change
CS_OH	Context switching overhead in cycles.	-

Data collected is divided into two categories based on the data scope. First category, system and runtime parameters, contains data which is relevant to the whole application execution. Execution frequency is defined for the whole trace if only single execution frequency is used. Otherwise execution frequencies are defined for each task. Profiling implementation is supplemented with a frequency table containing all available frequency levels in the system. Frequency table is used to define the accurate execution frequency. Total runtime equals the length of the trace or hyperperiod. All time values in profiling are defined by execution cycles and frequencies.

The other data category contains data specified for task found in the trace. Data for each task is stored into a data structure (Figure 15). Task WCEC equals the largest amount of cycles between a task-switch-in and the following task-switch-out. Each task instance is assumed to require its WCEC for completion and is defined by the number of task-switch-in events. A time interval value defined as the shortest interval from a task-switch-in event to the next can be used as a deadline value.



handle	count	events	WCEC	runtimes	time_window	periods
'00056010'	1161	2323×2 cell	11698896	1161×1 cell	1×3 cell	1160×1 cell
'0005A4A0'	4	9×2 cell	112176	4×1 cell	1×3 cell	3×1 cell
'0005C930'	1093	2187×2 cell	175765	1093×1 cell	1×3 cell	1092×1 cell

Figure 15. Example of data structure which holds task specific data collected from the execution trace.

4.5.1. *Profiling data visualization*

Although task dependencies are not profoundly studied in this work, the trace file can be used to create a simple visualization of the task execution order for each task. The order is visualized as a directed graph. For each task, the tasks which have been executed before and right after the task execution are shown. If the graph is simple, the order in which the tasks execute can be specified (Figure 16). For complex applications, it can be more beneficial to visualize one task at a time.

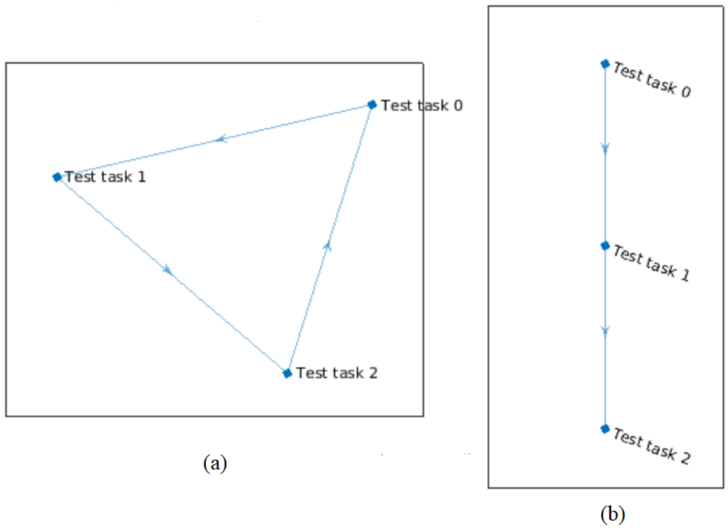


Figure 16. (a) Shown is an example of a case where the task execution order stays the same. (b) Preceding and following tasks can also be visualized for single tasks at a time which is extracted from (a).

Time view visualization and histograms of task execution characteristics give information on the amount of idle time and execution variation (Figure 17). Time view can be visualized in two ways. For each task, a separate bar visualizes the task execution also indicating which task is executed at given time in the trace file. Task execution and the amount of idle time can be also visualized by showing the division of execution time between idle and other executed tasks. Visualization does not specify which task is being executed any given time. Idle time is shown as gaps in bar graph. Execution time is shown on x-axis. Histograms are used to visualize variation between task inter-arrival times or execution cycles.

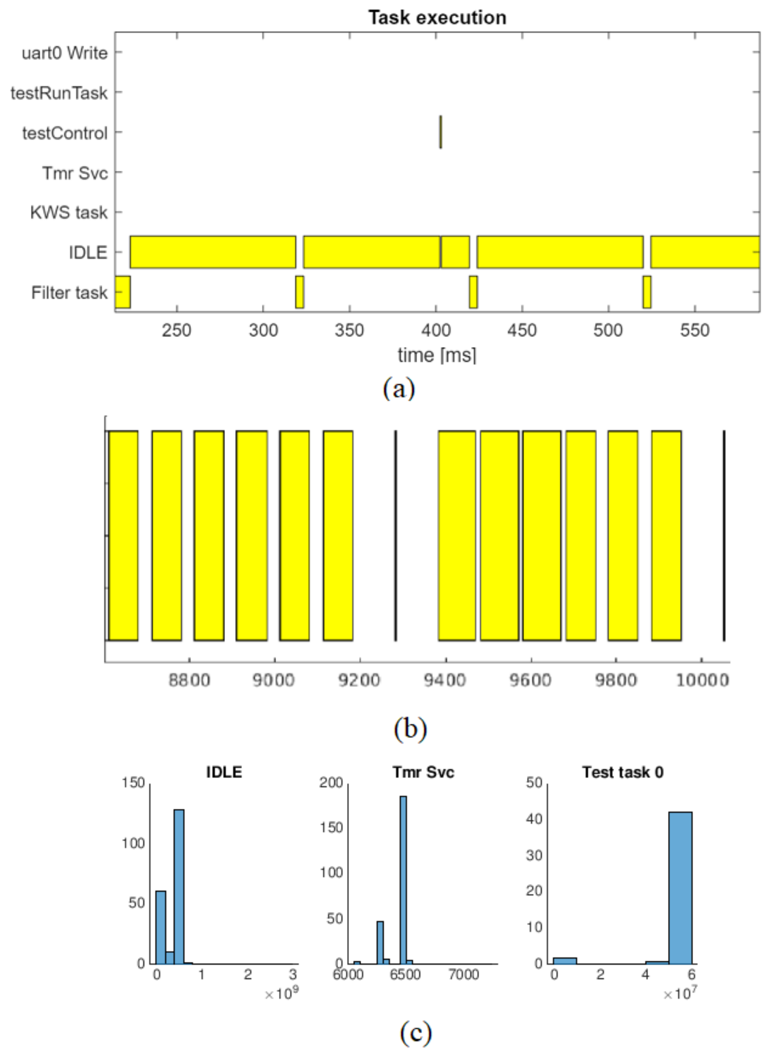


Figure 17. (a) General time view of task execution. (b) Task execution proportion to idle time. (c) Histograms of task inter-arrival times in cycles.

4.5.2. Target system

ARM Cortex M3 CPU is chosen as the target system. It is used as it is a pilot project for new low power technology. The target system is manufactured in 28 nm CMOS (Complementary Metal Oxide Semiconductor) technology and is capable of near-threshold to nominal voltage operation. The minimum energy point is proven to exist around 0.2 V – 0.4 V [56].

The target system includes on-chip low-dropout voltage regulator, PLL (phase-locked loop), stock memory, peripherals, and interfaces. The target system is shown in Figure 18 from [57] which provides detailed description of the target system. Timing safety margins are reduced by critical path monitoring and time-borrowing flip-flops. In case of timing events, time borrowing can be cancelled by dynamical clock stretching. [57]

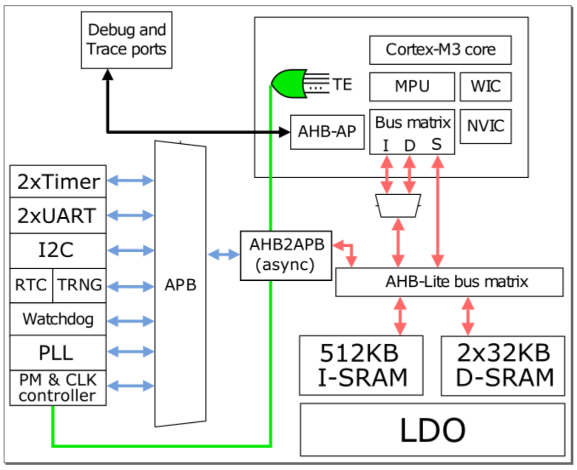


Figure 18. The target CPU system.

Overhead consists of the processor cycles that are not used to execute application code. Two kinds of overhead are defined for optimization. By including the overheads, task execution is modeled more realistically as many DVFS algorithms assume that overheads are negligible. Especially for hard deadlines, it is necessary to include overheads as otherwise the model is too idealistic.

These overheads are caused by OS, DVFS algorithm implementation and OP change times. The overhead values are defined for the described target system. The same system is used to execute the software and collect the measurement data.

Context switching overhead contains overhead caused by the OS. It includes saving the context of a task being switched out, scheduling and restoring context of a task being switched in. Context switch overhead is always present in the system and depends on the operating system and the processor used. When OP change related code is included and DVFS is used, the context switching overhead increases.

The context switching overhead values are measured for three cases and for both change directions i.e. levels of voltage and frequency values used for OPs. The test cases and corresponding values for single measurement are presented in Table 6. Values are defined by comparing the cycle count register value at known points where a task is known to switch out and switch in. The overhead value is the difference in execution cycles from a switch out to the next switch in.

Table 6. Context switch overhead values

Overhead	OP change direction	Cycles
FreeRTOS	-	992
FreeRTOS + DVFS code	Low to high	1721
	High to low	1872
FreeRTOS + OP change	Low to high	12889
	High to low	6455

FreeRTOS overhead values is for OS only. OP is not changed, and DVFS code implementation is not included in the application software. Thus, there is only overhead caused by OS. For FreeRTOS + DVFS measurement, the DVFS implementation is included but only single OP was forced to be used during execution i.e. voltage and frequency are not changed during execution and there is no settling delay. The values show that FreeRTOS and DVFS code overhead is approximately equal in both directions.

When OPs are actually changed i.e. voltage and frequency are changed when another task is switched in, it is seen that the overhead values differ between directions and are multiple times greater than previously presented values. The large difference between cycle count values can be explained with voltage settling delay. Settling delays are discussed in Subsection 2.2.2.

Because of the settling delays, the overhead caused by the voltage and frequency change is defined as a separate overhead value. It is excluded from the context switching overhead so that context switching overhead is defined in cycles. The values used in optimization for FreeRTOS on given target system are rounded upwards from the measurement values (Table 7).

Table 7. Context switch overhead values used for optimization

Overhead	OP change direction	Cycles
FreeRTOS	both	1000
FreeRTOS + DVFS code	both	2000

Each instance of task execution requires a context switch and the overhead value is defined in cycles. Thus, the overhead is included in the optimization model by adding the overhead cycles to the execution cycles required by a task. Voltage and frequency settling overhead cannot be accurately modeled in cycles. It is included in the model as a fixed time value. In Figure 19, the points where cycle count values are recorded during task execution are shown. Also, the different overheads are visualized.

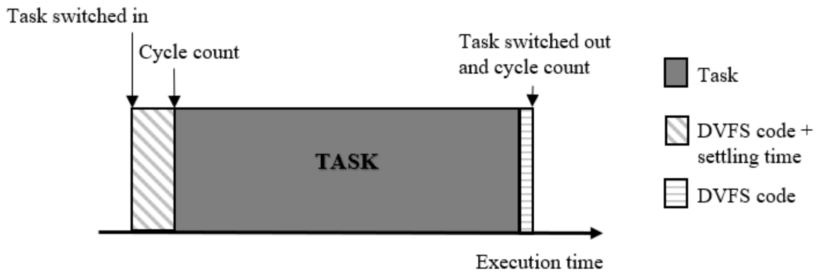


Figure 19. Points where cycle count values are recorded for task execution.

It is assumed that each task instance includes an OP change, excluding tasks which are not assigned any OP. Voltage and frequency change overhead is a major overhead in DVFS-enabled systems and cannot be neglected when execution frequencies are defined. The delays related to settling can be viewed as the cost of a context switch in.

Execution cycles needed to change voltage and frequency are measured by reading the cycle counter values before and after the respective functions (Table 8). The cycles are measured for case where OP is not changed and where OP is changed. The cycles required to execute the DVFS code without including any settling delay, i.e. the OP is not changed, is shown to be negligible. Thus, it is assumed that it is sufficient to include the change overhead as settling times. The overhead consists of two values: the time taken for frequency to lock and the time required for voltage to be set. It is seen from the measurements that the cycles needed for frequency to be changed is around 2000 cycles in both directions. For voltage change, greater number of cycles is required.

Table 8. Execution cycles taken by frequency and voltage change

Parameter	OP change direction	Frequency change	Voltage change
DVFS code	Low to high	21	16
	High to low	21	16
OP change	Low to high	1850	9023
	High to low	1995	2625

The voltage change overhead depends on the direction of the OP change and there is no way to define the change direction in the optimization phase unless the optimization is implemented iteratively. Thus, it is assumed that the overhead at each point where OP is changed, requires the worst-case overhead. A specified worst-case PLL frequency lock time is used for the frequency change overhead value. It is dependent on the reference frequency used for PLL. The voltage change ratio is defined to be 1 μs / 1 mV. The value for voltage change overhead is defined based on the difference between the lowest and the highest voltage level used.

4.5.3. User-defined data

Some information required in the optimization cannot be mined from the trace. These parameter values must be provided by the user and this requires detailed knowledge of the application software and the system used.

The minimum frequency depends on the system used. Minimum frequency value is set to the lowest frequency level that is available for execution. The minimum frequency is used with the task execution frequency to limit the frequency scaling. Thus, the execution frequency cannot be scaled under the minimum frequency level value.

A relative deadline is required to be defined for each task. Relative deadline is the time when a task must have completed all its jobs, measured from the release of the task for execution. It defines the range that the execution frequency can be scaled for a single task execution when other tasks and constraints are not considered. It is left to the user to verify that the set constraints and requirements are sufficient, and that the performance and deadline requirements are met.

The time amount that is reserved from the total available time, i.e. the time amount corresponding to the length of the trace section, must be defined to limit the time used for scaling and reserve time for other activity than task execution, for example interrupt processing. Task execution here consists the execution time for tasks included in the optimization. The amount of reserved guard time depends on the application and especially the activity and adequate value should be confirmed with testing. There are methods such as utilization bound which can be used to define a reference value for utilization [35]. Another method is the hyperbolic bound for RM scheduling [58].

When defining the reserved-time amount, the tasks which are left out of the optimization should be considered. In practice, due to the assumption that every task instance requires its WCEC, some slack arises when fewer cycles are required for execution. Idle task can have some low-level activities such as freeing memory, so some execution time for idle task needs to be guaranteed. It should be noted that the distribution of this reserved time cannot be defined, it is used to guarantee that this amount of time in the trace section is not spent executing application tasks in the optimization.

The reserved time from single task instance execution is defined in a similar way as for the total task execution. However, this time value is defined for each task separately and it is taken away either from the set deadline value. The reserved time is slack time from the perspective of optimization tasks. This should be considered when defining the value. If a task has a deadline which is always less that the task time interval value, slack arises because the frequency scaling will be limited by the deadline value. The value is also application dependent and to find a suitable value, the reserved time values should be tested.

The effect of reserving time from task execution in an ideal case is presented in Figure 20. Task frequency is scaled so that is requires the whole time that is available for execution when the overhead of OP switch and reserved time have been removed from the available time which here is the task deadline. During execution, this reserved time can be scattered between task execution if task execution is interrupted to perform some other activity, and task can still meet its deadline.

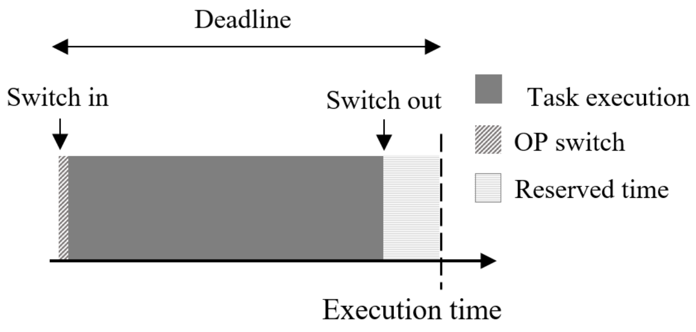


Figure 20. Effect of reserved time on available time.

The frequency range usable for a task can also be limited by setting scaling frequency bounds, otherwise values for current execution frequency and minimum frequency are used. Scaling limiting can be used if it is known that is it not beneficial to scale task frequency to the lowest level. Scaling factor limits for all tasks can be defined separately.

5. MEASUREMENTS

The frequency optimization testing is carried out in two phases. First, a speech recognition software execution is traced. A few different traces are used to define frequencies with different optimization constraints. The effect of constraint values to the optimization results is also examined. Averaged power measurements between a reference case and cases with optimized frequencies are used to estimate the change in energy consumption.

5.1. Tracing and measurement set-up

The target system is described in Subsection 4.5.2. The system includes on-chip PLL for clock generation and power management. The overhead values measured for the system are presented in Subsection 4.5.2. J-Link Base debug probe and RTT (Real-Time Transfer) are used to stream the execution trace data from the target to the host. SWD (Serial Wire Debug) is used as a target interface. SWD provides real-time access to memory [59]. RTT is used to stream the trace data from the target to the host computer without affecting the application execution.

RTT implementation includes a simple version of printf-function which is used to send event traces as formatted strings via RTT. RTTViewer is used to open connection between the target and J-Link and save the execution trace into a text file. A function which waits for input from the host is inserted into the beginning of main function in the application code before the scheduler is started and tasks are created. This allows the events to be collected from the beginning of the execution, collecting information for all task create events. [60]

The measurements were taken with a power monitoring unit and read within a periodic task. Due to the interference caused by the measurement task, the test traces were taken separately. The execution traces which are profiled are recorded first and executed using only single frequency values. Traces are processed as required and used in profiling. After the optimized frequencies are defined, respective voltage levels must be defined to form the voltage-frequency pairs. Because the measurement values cannot be timed into the execution trace precisely, the measurement values are averaged over the whole measurement trace. The average measurement values are defined for each used frequency which will be used to indicate the average power consumption for each OP.

5.2. Test application

The profiling and frequency optimization model are tested on ARM KWS (Keyword Spotting) software. KWS is a speech recognition software. A speech recognition software is a good example of an application with constrained power and as an always-on application. KSW is presented in more detail in [61]. KWS includes a task which extracts features from audio data and a classifier is included which is based on a neural network model.

Another task is added to the software to recognize human voice in the audio. The task includes a bandpass filter and thresholding for human speech detection. The filter task constantly filters the audio when available and when KWS task is not executed. The interval between instances is approximately 100 ms. If human speech is detected, KWS task is evoked to recognize the word. The relative deadline for both tasks is assumed to be 100 ms. The 100-millisecond deadline is set because after the time has passed, new data will be available for processing.

5.3. Frequency optimization

The execution traces which are used for optimization are called the reference traces. Measurements taken without the optimized frequencies are the reference values. The optimized traces are recorded with the optimized execution frequencies in use. The measurements are recorded for reference frequency and supply voltage and for each OP. The traces have varying amounts of no-voice audio and voiced audio. The traces are taken from different parts of the execution of the software, chosen by hand. The different traces are used to test how the trace section selection affects the optimization. Different values used in the optimization for the same trace section can be used to show the effect on the optimization results.

5.3.1. Trace 1

The first section used for optimization is shown in Figure 21. The length of the trace sections is approximately 2321 milliseconds. OP switching is not used or included, and the application is executed with a single frequency. It is clearly visible in the picture that there is a lot of idle time between application task execution. This section of the trace was used for three optimization cases where the context switching overhead and requirements for the reserved time values are varied. All variable values used in the optimization for the three cases are presented in Appendix 2 - Appendix 4.

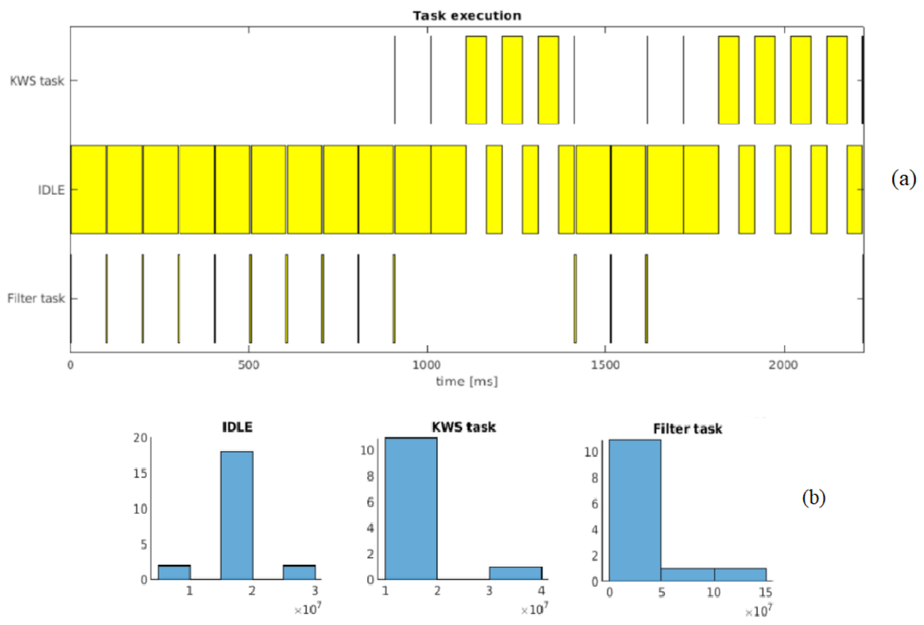


Figure 21. (a) Time view of trace 1. (b) Task execution intervals in cycles.

Figure 21 also shows the variation of task execution intervals in the trace. A single frequency was used in the optimization and the execution intervals are expressed in execution cycles. It is seen that the tasks are not periodic. However, it is seen that one of the tasks is executing, depending whether voice is detected or not, rather periodically. How the audio data is handled explains the variation in execution intervals. KWS task execution is triggered at 900 ms time point. Task periods are not used define task execution in the optimization. Instead, task instance count is used to define the total execution time required by a task.

The constraint input values are presented in Table 9. The reserved time value was used for both tasks. As the same trace section is used for all cases, the values collected from the trace are same for all cases. Context switching overhead is assumed to be zero and there is no requirement for any idle time in Ideal case. This means that the frequency can be scaled the most within the limits set by trace data and deadlines. Overhead-only takes into consideration the cycle amount needed by each context switch and Restricted case requires some time off from the optimization task execution.

Table 9. Overhead and constraint values for the three optimization cases

Optimization case	Constraints		
	Context switch overhead	Total reserved time	Reserved time in task's time window
Ideal	0 cycles	0 ms	0 ms
Overhead-only	2000 cycles	0 ms	0 ms
Restricted	2000 cycles	200 ms	5 ms

The optimization results for KWS and filter task for these three cases are presented in Table 10. The scaling factor is the result from the optimization. It is seen that the filter task is scaled to the lowest frequency level available in all cases. Table 10 also shows the actual frequencies that were used for the given optimized frequencies and the corresponding supply voltages.

Table 10. Optimization results and runtimes after optimization for the three cases

	Case	KWS	Filter task
Scaling factor	Ideal	1.7875	15.104
	Overhead-only	1.7872	15.1040
	Restricted	1.5551	15.1040
Scaled frequency [kHz]	Ideal	101 397.482	12 000
	Overhead-only	101 414	12 000
	Restricted	116 551	12 000
Actual frequency [kHz]	Ideal, Overhead-only	102 400	12 037
	Restricted	117 760	12 037
Corresponding voltage [mV]	Ideal, Overhead-only	528	420
	Restricted	590	420
Instance runtime [ms]	Ideal, Overhead-only	55.66	4.63
	Restricted	55.664	4.633
Instance runtime after optimization [ms]	Ideal, Overhead-only	98.526	69.766
	Restricted	85.6747	69.766
Instance runtime after optimization (ideal) [ms]	Ideal, Overhead-only	99.483	69.981
	Restricted	86.5634	69.981

Actual execution frequency for KWS task increased 0.99 % from the optimal frequencies due to the implementation in ideal case. The magnitude of the difference is as expected as frequency steps are larger in higher frequencies. In overhead-only case the frequency for KWS task is increased 0.97 %. Ideal and overhead-only cases result in the same actual execution frequencies even though the constraints were different. In Restricted case, KWS task is assigned a higher frequency than in the previous cases. The frequency is increased 1.04 % due to the implementation. The increase for filter task was 0.31 % for all cases.

In Table 10, shown are the estimated worst-case execution times of the tasks. Execution times are defined with the optimization result frequency and the closest, higher frequency level that is actually used. The execution times are estimates based on the WCEC value for a task. In Ideal and Overhead-only cases, the theoretical execution time for KWS task with the exact optimized frequency leaves just enough time for OP switch. The execution time for KWS task increases 77.0 % and 1406.8 % for filter task. In Restricted case the execution time is estimated to increase 53.91 % for KWS task.

The difference between Restricted case and the other two test cases was that case three included a lot of reserved time both from total time and task time window. The total reserved time was about 8.6 % of the total time available. This is seen as higher frequency for KWS task. The execution time of filter task is bounded by the lowest frequency level. It is seen that the filter task is still scaled to the lowest frequency level and fulfills the 5-millisecond reserved time requirement.

The frequency deviation from the optimization result causes some more possible idle time in the application execution. The amount is visible in the difference of the estimated runtimes for a task instance assuming that all task instances require their WCEC (Table 10).

5.3.2. Trace 2

The second trace is longer than the previously used trace section and contains multiple executions for word classification task and filter task (Figure 22). The length of the trace is approximately 34300 milliseconds. The context switching overhead is included but no reserved time required in the optimization. The rest of the values used in the optimization are presented in Appendix 5.

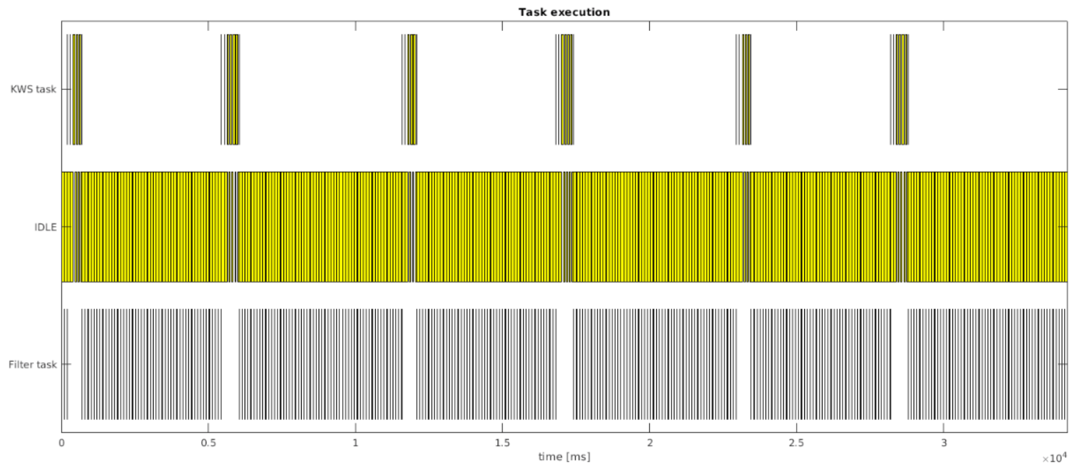


Figure 22. Task execution in trace 2.

It is seen that the final execution frequencies correspond to the values defined for cases 1 and 2 in trace 1 optimization. The frequency for filter task increases 0.31 % and 0.97 % for KWS due to the implementation. The frequency values and estimated instance runtimes are presented in Table 11. Execution time for KWS task increases 77.00 % and 1405.8 % for filter task by estimate for worst-case instance.

Table 11. Optimization results and runtimes for trace 2

	KWS	Filter task
Scaling factor	1.787	15.104
Scaled frequency [kHz]	101416.202	12 000
Actual frequency [kHz]	102 400	12 037
Corresponding voltage [mV]	528	420
Instance runtime [ms]	55.66469	4.63335
Instance runtime after optimization [ms]	98.52651	69.767051
Instance runtime after optimization (ideal) [ms]	99.4822	69.9821

5.3.3. Trace 3

The third trace section used for optimization is short and resembles the trace 1 (Figure 23). Trace section length is approximately 2.3 seconds. The optimization for trace 3 contained only overhead caused by FreeRTOS as the context switching overhead value. The OP switching was used during tracing but a single OP was forced to be used. The values used in the optimization are presented in Appendix 6.

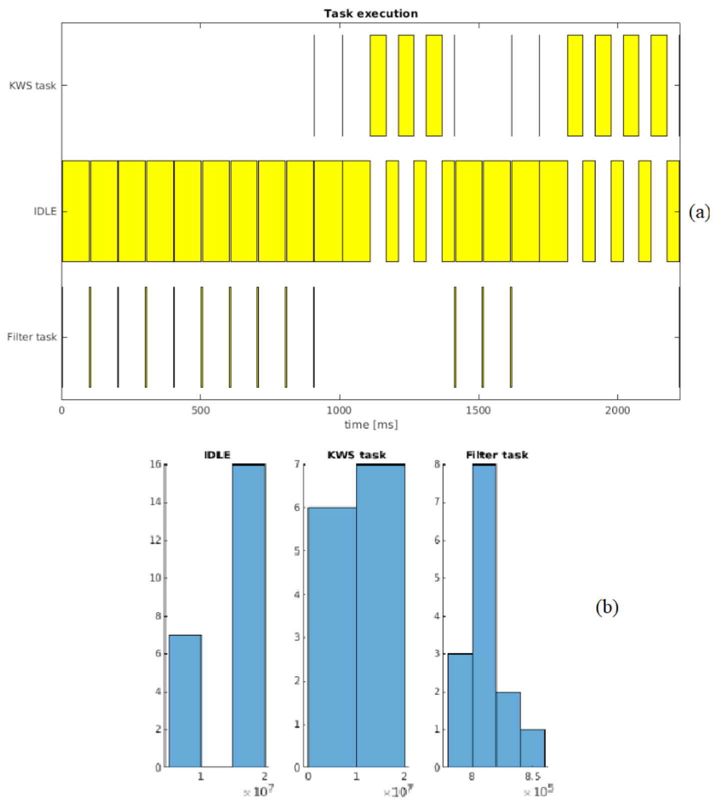


Figure 23. (a) Task execution in trace 3. (b) Task execution cycles variation.

It is clearly seen in the time view visualization that the KWS task execution length has large variation between shortest and longest execution. The variation in filter task execution is smaller which can also be seen from the histogram (Figure 23). Nearly half of the KWS task instances require small amount of execution cycles in selected trace section. When these kind of task instances occur in the execution with optimized frequencies, a lot of dynamic slack is bound to occur. For filter task, the amount of slack should be smaller as the range of execution cycles is smaller.

There was no requirement for reserved time. It is seen that the resulting frequencies (Table 12) are the same that have been defined in the optimization cases presented earlier. Frequency for filter task increases 0.31 % and 0.98 % for KWS from the optimized frequency values. The estimates for task execution length are given in Table 12. Execution time is estimated to increase 77.0 % for KWS and 1405.8 % for filter task.

Table 12. Optimization results and runtimes for trace 3

	KWS	Filter task
Scaling factor	1.7872545319731	15.104
Scaled frequency [kHz]	101 408.828	12 000
Actual frequency [kHz]	102 400	12 037
Corresponding voltage [mV]	528	420
Instance runtime [ms]	55.66758	4.636757
Instance runtime after optimization [ms]	98.53162	69.81831
Instance runtime after optimization (ideal) [ms]	99.49467	70.033583

5.3.4. Trace 4

The last trace which was optimized is shown in Figure 24. The trace was recorded with OP in use but forced to use a single OP. The optimization included context switching overhead for FreeRTOS. Rest of the optimization values are presented in Appendix 7. The length of the trace is approximately 909 milliseconds. It is seen that the trace mostly contains idle and KWS task execution with long execution times.

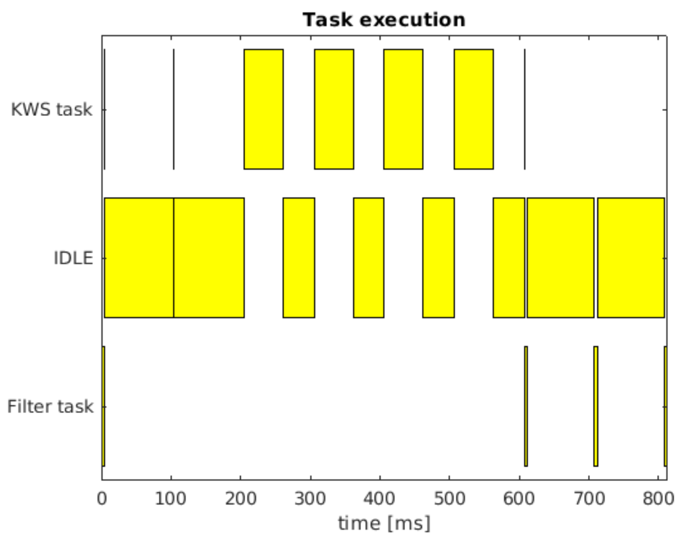


Figure 24. Task execution for trace 4.

Table 13 shows that the frequency for filter task is again scaled to the lowest level. The frequency for KWS task is scaled to around 113 MHz although there were no requirements for reserved time. Due to the implementation the frequency of KWS task increases 1.2 %. The estimated execution time for filter task stays around the same value as in previous optimization cases (Table 13). The execution time for KWS task is a lot lower than most optimization cases and around the same level with trace 1 case 3 optimization.

Table 13. Optimization results and runtimes for trace 4

	KWS	Filter task
Scaling factor	1.59937853676076	15.1040
Scaled frequency [kHz]	113 324	12 000
Actual frequency [kHz]	114 688	12 037
Corresponding voltage [mV]	580	420
Instance runtime [ms]	55.626826	4.635648
Instance runtime after optimization [ms]	87.91025	69.8016
Instance runtime after optimization (ideal) [ms]	88.96836	70.01683

5.4. Optimization results

The optimization results lead to a few different OP value pairs to be defined. The filter task was scaled to the lowest frequency level in all optimization cases. The lowest level is the same that is assigned to the idle task in the optimized execution. This corresponds to 12 MHz and 420 mV OP values. The rest of the frequencies are presented in Table 14. Requested frequencies which are assigned to an OP and resulting execution frequencies are presented also with assigned voltage levels. Table 14 shows that actually only three different execution frequencies are used for KWS task. This is because of the frequency rounding to an integer megahertz value.

Table 14. Final execution frequencies and actual execution frequencies used with respective voltage levels

Frequency [kHz]	Requested frequency [MHz]	Actual execution frequency [kHz]	Corresponding voltage [mV]
12 000	12 000	12 037	420
101 398	102 000	102 400	528
101 416			
101 417			
101 409			
113 324	114 000	114 688	580
116 551	117 000	117 760	590

The time view of task execution with optimized frequencies is pictured in Figure 25. Idle task and filter task are executed with 12 MHz and KWS with 102 MHz. The effect of frequency scaling is clearly visible in the proportion of idle task and application task execution compared to the time view figures presented in Subsection 5.3. 102 MHz was the lowest frequency that was assigned to KWS task. It also shows how there is always some idle time left.

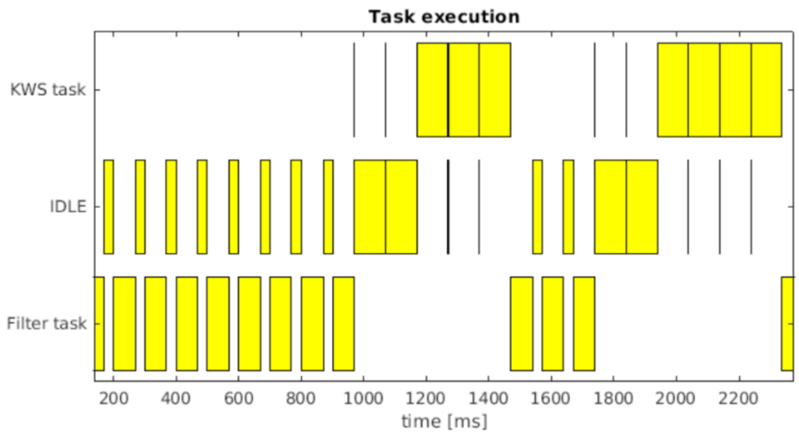


Figure 25. Time view of how application task execution is prolonged with DVFS. 102 MHz execution frequency for KWS is shown.

The filter task frequency scaling was bounded by the lowest frequency level. Because of this and the fact that the audio is processed in approximately 100 millisecond sections, the filter task execution time stays quite constant. The execution time of KWS task varies more. Around 1000 millisecond mark, KWS task is executed only for a short time. This leads to idle task execution. This is where it would be beneficial if the DVFS decision were done online because this slack that rises because of runtime variation can’t be utilized when offline DVFS algorithm is used.

5.4.1. Power measurement results

Measurements were recorded with a periodic measurement task run with 200 milliseconds intervals. Due to the interference caused by measurement task the execution traces and traces including measurement values were taken separately. Average measurement values are used to define power consumption value at each OP. Because there are only three different pairs of OP values, three different measurements were done. The reference measurement which is used to compare the total energy consumption of a task after the optimization was executed with a single frequency. The execution frequency was 181 248 kHz and voltage 903 mV.

The benefits of lowering the supply voltage and execution frequency are seen in the long run when it is seen how the energy consumption has changed. The energy consumption depends on the application activity and it is clear that if the consumption was defined for a certain section of the execution, the section selection would affect the values a lot. Thus, we use the energy consumption of the application task KWS to model the effect of DVFS. For the optimized cases, task execution time is defined based on the WCEC value used in the optimization. To make the results more comparable, a single WCEC value is used. Energy consumption for a task execution is defined based on the power measurement values and the execution time. The reference value is defined for the same execution time and measurement values from the reference execution trace. This means that the idle and filter task energy consumption is not considered in the optimized measurements.

The measurement values for each OP are presented in Table 15. The first row corresponds to the reference measurements and the rest three are the OP defined based on the optimization result. Table 15 includes power values which are the averaged values of power measurements over the execution of KWS. The measured power values are used to define the energy consumption within given time.

Table 15. Power average measurements for KWS task and estimated energy consumptions for task execution

Execution frequency [kHz]	Execution time of KWS [ms]	Assigned voltage [mV]	Measured power [µW]	Energy consumption [µJ]
181 248	55.67	903	2809	156.4
102 400	98.53	528	640,0	63.1
114 688	87.97	580	863,6	76.0
117 760	85.68	590	914,0	78.3

The energy consumption is defined for each OP used. It is assumed that the execution time scales linearly, and the energy consumption is defined accordingly. For the reference measurement which corresponds to the first row, the value includes only the KWS task execution time. The estimated values are presented in Table 15. The execution cycle count of 10089638 is use as the assumption WCEC for KWS task to make the comparison between different OPs easier. It is the largest WCEC value that was used in the optimization. With 102 MHz the execution time of KWS increases 77.0 % and energy consumption decreases 59.7 % compared to the reference case. Increase in execution time is 58.0 % with 114 MHz frequency and energy consumption decreases 51.4 %. For the 117 MHz execution time increases 53.9 % and energy consumption decreases 49.9 %.

The energy consumption for the total length of optimized KWS task execution is calculated in the reference case. Energy consumption in the reference case and for each OP used for KWS task (Table 16) is compared separately and the time which is used to define the total energy consumption is different in each case. This includes the execution of KWS task and idle task. The energy consumption for the time of KWS task execution with 102 MHz decreases 77.2 % compared to the energy consumption in reference case. For the time task KWS task is executed with 114 MHz the energy consumption decreases 69.2 % and for the 117 MHz KWS execution energy consumption decreases 67.4 % compared to the energy consumption in that time with the reference case.

Table 16. Energy consumption in reference case for the time of KWS task execution in optimized cases

Frequency [kHz]	Execution time [ms]	Energy consumption [µJ]
181 248	98.53	276.8
181 248	87.97	247.1
181 248	85.68	240.1

5.4.2. Optimization discussion

It is seen that varying trace lengths and constraints resulted in little variation in final execution frequencies. Some remarks can be made about the application. With only two tasks included in optimization, it is on the simpler side. It is known that the tasks have a dependency. They are not executed periodically, more like in turns. There is some variation between execution cycles of task instances, especially with KWS task. However, it was out of scope to design the application or optimize it for the processor architecture used (KWS was supplied by the processor IP manufacturer).

The effect of overheads, constraints and section selection were tested. Ideally, the frequency is scaled so that there is just enough time to complete all tasks. Due to this and the assumption that there is dynamic slack, reserved time was not requested in most of the optimization cases.

The results show that actual execution frequencies do not have much variation. In this case, the context switch overhead is relatively small compared to task execution cycles. The effect of including overhead is accentuated when task execution cycles decrease and the percentage of overhead of total cycles increases. Therefore, context switch overhead should be included in the optimization although it can be compensated by the frequency deviation. Results also show that in this case the reserved time requirement affects the optimization result more than the included overhead causing increase in execution frequency for KWS.

Scaling of the filter task was limited by the lowest frequency level. For KWS task, the optimized frequency varied. It was seen that even if the optimized execution frequencies were different, defined with different constraints, the final execution frequencies were the same. Small variation in the constraints did not affect the final execution frequencies. This is due to the discrete frequency levels and that the execution characteristics of the application remain the same.

6. DISCUSSION

The application characteristics and requirements need to be considered when choosing the most suitable power management technique. For other application might benefit more of sleep states while other applications are more suitable for DVFS. The impact on power consumption and responsiveness needs to be considered. Low power techniques can also affect the system reliability.

Sleep states can be applied if idle periods are known and longer wakeup delays do not cause performance degradation. DVFS does not introduce long wakeup delays. However, changing OPs introduces timing delays and increase run time. This can cause deadline misses if frequencies are defined incorrectly and therefore knowledge of task and application characteristics is required. If performance can be kept at acceptable level and a suitable algorithm is used for DVFS, it can reduce dynamic energy.

The quintessential question with DVFS is how the execution frequencies can be defined so that system still meets its processing requirements. Many DVFS algorithms were discussed before and all algorithms have their limitations. There is no single algorithm which is optimal for all applications. Traditional algorithms are mostly developed for specific scheduler and make assumption about the task set. These algorithms cannot be applied to any application. Machine learning has been used to overcome the problem being able to adapt to runtime variation and utilizing slack more efficiently, like in [45]. However, the proposed frequency optimization is developed for offline inter-task DVFS. Offline algorithms are simple and thus introduce low overhead.

The proposed optimization method takes a different approach on frequency optimization by analysing and profiling software execution trace. Trace data is used to form a model which is used for frequency optimization. This way any application can be profiled. It provides a more general solutions on how the execution frequencies are defined. This way frequency optimization can be used to estimate the energy consumption savings that can be achieved on given platform when DVFS is used and with any OS with minimal code instrumentation. Although the core idea is to scale the execution frequency as low as possible, user can easily change constraint values to limit the scaling as desired. It is simple to explore the effect of system overhead and timing constraint values to the optimization results.

6.1. Limitations

The generalization of the model and offline DVFS can also lead to suboptimal results in some applications. As there is no adaptation to runtime variation and frequency decisions are done based on worst-case assumption, frequency can be set to an unnecessarily high level. This leads to suboptimal result if a task rarely requires its WCEC and dynamic slack cannot be utilized. WCEC values must be used to ensure deadlines.

The scheduling algorithm is also not considered in the optimization model. It is intended that scheduling is not affected by frequency scaling meaning that a task can have time available for scheduling at most the time amount of the shortest time window value. However, it is left to the user to define which value is used as deadline value to limit the scaling. If tasks use close to their WCEC most of the time and the execution pattern is deterministic, better optimization results can be achieved with the optimization model. Thus, the use cases for this DVFS algorithm are always-on applications, applications which have predictable execution pattern and low activity leaving enough slack for frequency to be scaled notably.

DVFS algorithms use either discrete or continuous frequencies even though only discrete frequency levels are available in systems. Continuous frequencies are used in the optimization model to simplify the problem. Efficient DVFS implementation requires that a variety of frequency levels are available. The assumption of continuous frequencies causes the actual execution frequencies to deviate from the optimized values. The actual execution frequency should be set to a value as close as possible to the optimization value and there should be also low voltage levels available to match the execution frequencies to gain benefit of DVFS. The deviation from the original optimized frequency value might not be that significant itself, although it varies between systems. However, the voltage is set accordingly, and increase results in greater energy consumption. In the long run, the difference might become significant. The overall impact of DVFS is hard to estimate with short measurements.

On the test system used, it was known that the step size between frequency levels depended on the frequency range. Frequency deviation by DVFS control could be avoided by using the knowledge of which frequencies are produced by which frequency assignments or slightly altering the implementation. Also, this impact can be considered minimal as most of the total amount of frequency deviation is probably caused by the lack of many frequency levels. The total amount of deviation is defined case by case as it depends on the optimized frequency values and the system used. It can be thought that the frequency deviation in the test cases were minimal compared to the whole frequency range that was in use.

The number of OPs is not limited meaning that there can be as many different execution frequencies defined as there are tasks in optimization. As every OP change introduces overhead, constant changing can result in increased power consumption and overhead. If the number of OPs is limited, the number of OP changes would decrease. How can the optimal number of OPs be defined? A possible solution could be to group similar OP to their highest values i.e. all tasks which have an OP close to each other would be assigned the highest OP among the group. This could be beneficial if there are a lot of different tasks with different OPs.

The cost of OP change is only included as timing overhead in the model. It is included to ensure that deadlines are kept even with overhead caused by OP change. The effect of frequent OP changes was discussed in the previous paragraph. Thus, the power consumption impact of changes should be considered. System power model and the impact of OP changes need to be integrated into the frequency optimization.

As energy consumption does not only depend on the efficiency of the DVFS algorithm, it is not easy to compare different algorithms. As was just discussed, the energy consumption reduction also depends on how low the voltage can be set and how accurately the execution frequencies can be set to match the execution frequencies if continuous frequency range is used. The cost of OP change on the given platform must also be considered. The efficiency also depends on the application. Thus, the limitations of the used DVFS algorithm need to be acknowledged.

The complexity of the frequency optimization problem increases as more variables are included in the model. However, this can lead to better optimization results for applications that match the model. An optimization tool for any application should have a power model of the system used and be able to estimate whether offline algorithm is adequate or would online algorithm be more beneficial. The requirement for intra-task OP changes should be evaluated too. How much overhead is acceptable and does not cause the energy consumption to increase?

6.2. Use considerations and improvements

User should have some knowledge of the application for better results. Decisions such as which tasks to include and required slack times affect the optimization result. The method can be developed further. A more detailed tracing and execution modelling is required and the relation between execution frequencies and actual execution times for tasks should be included.

6.2.1. Tasks suitable for optimization

Applying DVFS blindly can lead to higher energy consumption. Some tasks should be left out of optimization to avoid constantly changing the OP. Constant OP changing defers application execution and can lead to higher energy consumption. Generally, short tasks which execute frequently should be left out of the optimization. This is especially the case when settling times are long. OP change should not take longer than the execution of the context's jobs e.g. it is not reasonable to change OP before executing interrupt handlers.

6.2.2. Required slack time

The reserved time requirement for tasks affects the optimization. Needed time depends on the application: how much of the activity is not included in the optimization. If the trace section starts with an idle period, the time needs to be included in the reserved time requirement. The variation of task execution lengths can be used to assess if reserved time requirement can be decreased as dynamic slack arises from variation. This is especially the case if task time window is used as a deadline. For task-specific requirements, the impact of possible pre-emptions needs to be considered. There should always be enough time for time-critical tasks to complete execution before deadlines.

Utilization bounds can be used as an estimate for the total amount of needed reserved time. These should be used as a guideline as not all tasks are necessarily included in the optimization and tasks are not periodic which is usually assumed. Defined bounds can be pessimistic and result in under-scaling.

In practice, the frequency is scaled as low as the deadline allows with no time reservations. If the total runtime is less than required time for task execution with low frequencies, it becomes the limiting factor. The problem with total reserved time is that the time allocation cannot be defined which is done with task-specific reserved time. Some tasks may not use the whole time window even with the lowest frequency and reserved time is not required. Values for total reserved time and for each task's time window need to be chosen and tested together to find suitable values.

6.2.3. Task execution, deadline misses and prioritization

Relative deadline defined the time limit by which task must finish all its jobs. How deadline misses are detected? Do they need to be detected if they are not fatal? Calculating execution times and comparing to deadlines during execution is not reasonable. The execution trace could be used to detect deadline misses. It is assumed that the user is aware of the consequences of deadline misses. The severity of a deadline miss depends on the application and the task. Time view visualization can be used to inspect execution patterns. Anomalies can be visible but slight deadline misses that do not cause failure can go unnoticed. Timings between trace events can be approximately defined with the current trace information. Precise timing would require accurate modeling of the system's voltage and frequency settling times. Even if voltage and frequency settling times are not modeled accurately, the worst-case assumption can be used for deadline miss inspection, although it would be a pessimistic approach.

An additional trace print can be inserted into the switch in macro before the DVFS callback function. Jointly with a task switch in print, it is used to define the section for OP change. Now the cycles between a switch out and the next switch in are assumed to be executed with the same frequency as the task which was switched out. Although trace prints defer application execution, the impact is probably lesser than constantly calculating runtimes. Total execution time for a task instance could be then defined by comparing the total task execution time and the OP change time to the deadline value.

Another question raises when task deadline misses are defined. How is it known if a task has finished its jobs? It has been assumed that tasks are executed without pre-emption and switched out once execution has reached the end of a task. Detailed knowledge of task implementation and additional code instrumentation is required if a trace print for task completion is added. This way the total task execution time could be defined combining execution times between the first switch in and the following task completion trace event.

In addition to relative deadline, absolute deadline could be checked from execution trace. Absolute deadline is the time when a task must be finished from the timepoint when it entered the ready state. Detecting absolute deadline misses could be implemented the same way as relative deadlines. For example, FreeRTOS provides a trace macro for tasks entering the ready state. The knowledge could also be used in optimization if a task has absolute deadline and it is seen that the task does not immediately get execution time which eventually can result in a deadline miss.

The assumption is that tasks have hard deadlines that should always be met. But tasks do not always have hard deadlines. This leaves room for optimization. Tasks which have soft deadlines could be assigned lower frequencies than their WCEC defines. An example of a potential task is a task which only occasionally requires execution cycles close to WCEC value and a deadline miss is not fatal. Instead of using WCEC in optimization, median or mean execution cycle value could be used in optimization. But if task execution is prolonged, it must be ensured that tasks with hard deadlines get execution time. This brings us the question if task priorities and OP levels have or should have a dependency.

Task prioritization should be done carefully when fixed values are used and when prioritization is done by the developer. Task priorities are not considered in the frequency optimization model. It can be thought that tasks with higher priorities require to be executed more urgently. Thus, should higher priority values correspond to higher OP values? Each task and application should be inspected as separate cases, but most critical tasks should be executed briefly and have a high priority to be able to do so. For tasks that require brief execution, frequency scaling can be limited by setting the scaling factor limits accordingly.

6.2.4. Additional profiling and detailed modelling

Complex applications consist of tasks which have interdependencies. One step to automate trace analysis and profiling could be to analyze task dependencies from the trace. If task communication events are not traced, task dependencies could be mined from the execution trace using pattern recognition. Task dependencies can be seen as repeating patterns in the trace [62]. If tasks are known to have a dependency, a possibility to define a joint deadline for these tasks could be added to the optimization. In a similar manner, pattern recognition could be utilized in suitable trace section selection. This would also require more detailed tracing, including information of task completions.

Static code analysis is another technique which could be utilized for task execution path and WCEC analysis. Variation in task execution paths could be used to define the least amount of execution cycles that is required for task completion. This knowledge of range for execution cycles is used to estimate the number of times a task was executed in the trace section if there are no events to define task completion. Acknowledging pre-emptions requires that the estimated number of pre-emptions and added overheads are included in optimization for each task. Frequency optimization becomes inevitably more complex as task execution is modelled more accurately.

More detailed modelling requires additional trace events. Even if only minimal tracing information is collected, tracing affects the application execution. The effect of selected scheduling algorithm is also not considered. Thus, regular testing should be carried out when scaled frequencies are applied. The required performance needs to be verified by the user. In practice, frequency scaling should be applied in the software development phase. The joint effect of scheduling algorithm, task prioritization and frequency scaling could be analysed and utilized in software development. For example, costly jobs which would be beneficial to execute with high frequency can be implemented as separate tasks.

6.2.5. Relation between execution frequency and execution time

The effect of different execution frequencies to task execution times should be studied. Although execution frequency is only changed at task boundaries, some insight on task execution would be useful in optimization. The assumption is that the execution time of a task scales linearly with its execution frequency. As this is not always the case, analysis on the execution inside a task could be used to group tasks based on how their execution time is affected by frequency scaling.

Tasks can be divided into CPU-bound tasks, where execution time is limited by the CPU frequency, and memory-bound tasks, if majority of execution time consists of memory accesses. If a task is memory-bound, execution frequency can be scaled to a low frequency level causing only minor effect on the execution time [63]. Some tasks spend time in I/O operations and frequency scaling can be limited by the subsystem. The power consumption for these subsystems should be modelled and considered in frequency scaling. It is not beneficial to scale the execution frequency of a task if it requires an expensive subsystem to be powered on for longer period if this results in greater total power consumption.

6.3. Conclusion

This leads to the conclusion that only scaling execution frequency might not always lead to lower power consumption. Processor power consumption can be reduced with DVFS, but it is not solely enough. Low power is achieved by combining different techniques at different abstraction layers. The impact of each component must be considered and techniques for low power must be applied at all design abstraction layers. Software must be also designed with energy efficiency in mind and to match the underlying hardware. Techniques should not be applied blindly. Tools are required which provide models how different design techniques and optimizations impact the power consumption and if certain techniques contradict and result in higher total power consumption.

7. SUMMARY

The objective of this thesis was to develop a method for DVFS frequency optimization using software analysis and profiling for near-threshold processor. The target system included an inter-task DVFS software. The method was tested with a speech recognition software.

It was shown that software execution can be modeled with only a few selected trace events. The trace data in combination with timing constraints and system-specific worst-case overhead values are used to form an optimization model. The method can easily be used to test the effect of constraints on the resulting execution frequencies to find suitable values. The resulting frequencies fulfill the set timing constraints. The results obtained from power measurements showcase the effect of DVFS on energy consumption of the near-threshold processor. Energy consumption is decreased using optimized execution frequencies and respective voltages for the given application.

As offline algorithm is not able to adapt to runtime variation, it is most suitable for applications with predictable execution patterns and tasks which regularly require execution cycles close to their WCEC. The reduction in energy consumption with DVFS also depends on the system.

The aspects of energy efficiency and low energy consumption must be considered at all stages of system design. Tools are required to model and give feedback on the effects of design decisions to the system energy consumption. In addition to power-aware hardware design techniques, software needs to be optimized for energy efficiency. Total system optimization is required. Power management can be applied to reduce power consumption even more. However, suitable technique needs to be selected based on application characteristics.

8. REFERENCES

[1] Kallimani R & Rasane K (2015) A survey of techniques for power management in embedded systems. International Journal of Emerging Technology in Computer Science & Electronics 14(2): 461-464.

[2] Oshana R (2012) DSP for Embedded and Real-Time Systems. Oxford, UK, Newnes.

[3] Durrani Y (2016) Low-power integrated circuit design optimization approach. UET Taxila Technical Journal 21(1): 32-42.

[4] Chatzigeorgiou A & Stephanides G (2002) Energy Issues in Software Design of Embedded Systems. 2nd WSEAS International Conference on Applied Informatics. Rethymnon, Greece, 1-6.

[5] Electronics Weekly (2009) Use the RTOS to save power. URL: www.electronicsweekly.com/market-sectors/embedded-systems/use-the-rtos-to-save-power-2009-11/. Accessed 3.2.2020.

[6] Biswas A (2020) CMOS Inverter - Power and Energy Consumption. URL: www.technobyte.org/cmos-inverter-power-energy-consumption/. Accessed 25.8.2020.

[7] Beningo J (2019) ARM Cortex-M low-power mode fundamentals. URL: www.embedded.com/arm-cortex-m-low-power-mode-fundamentals/. Accessed 3.2.2020.

[8] Yiu J (2014) The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors. Oxford, United Kingdom, Elsevier Science & Technology.

[9] Walls C (2015) Power management in embedded software. URL: www.embedded.com/power-management-in-embedded-software/. Accessed 9.3.2020.

[10] Beningo J (2019) How low (power) can you go?. URL: www.embedded.com/how-low-power-can-you-go/. Accessed 2.3.2020.

[11] Lattice Semiconductor Corporation (2005) Dynamic power management in an embedded system. URL: www.latticesemi.com/-/media/LatticeSemi/Documents/WhitePapers/AG/DynamicPowerManagementinanEmbeddedSystem.ashx?document_id=9552.

[12] Iniewski K (2010) CMOS Processors and Memories. Dordrecht, Netherlands, Springer.

[13] Benson M (2014) The Art of Software Thermal Management for Embedded Systems. New York, United States, Springer.

[14] Thati V, Vankeirsbilck J, Pissoort D & Boydens J (2019) Hybrid Technique for Soft Error Detection in Dependable Embedded Software: A First Experiment. International Scientific Conference Electronics. Sozopol, Bulgaria, 1-4. DOI: http://dx.doi.org/10.1109/ET.2019.8878497.

[15] Thati V, Vankeirsbilck J, Penneman N & Pissoort D (2018) CDFEDT: Comparison of Data Flow Error Detection Techniques in Embedded Systems: An Empirical Study. Proceedings of the 13th International Conference on Availability, Reliability and Security. Hamburg, Germany, 1-9. DOI: http://dx.doi.org/10.1145/3230833.3230854.

[16] Amara A, Ea T & Belleville M (2010) Emerging Technologies and Circuits. Heidelberg, Germany, Springer.

[17] Henkel J, Pagani S, Amrouch H, Bauer L & Samie F (2017) Ultra-Low Power and Dependability for IoT Devices. Design, Automation & Test in Europe Conference & Exhibition. Lausanne, Switzerland, 954-959. DOI: http://dx.doi.org/10.23919/DATE.2017.7927129.

[18] Castano V & Schagaev I (2015) Resilient Computer System Design. Cham, Switzerland, Springer.

[19] EEMBC (2019) Introducing the ULPMark™-CoreMark® Benchmark. URL: www.eembc.org/ulpmark/. Accessed 24.2.2020.

[20] Oshana R & Kraeling M (2013) Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications. Oxford, United Kingdom, Elsevier Science & Technology.

[21] Rahma M & Anis M (2013) Nanometer Variation-Tolerant SRAM: Circuits and Statistical Design for Yield. New York, United States, Springer-Verlag.

[22] Borkar S (1999) Design challenges of technology scaling. IEEE Micro 19(4): 23-29. DOI: http://dx.doi.org/10.1109/40.782564.

[23] Reynders N & Dehaene W (2015) Ultra-Low-Voltage Design of Energy-Efficient Digital Circuits. Cham, Switzerland, Springer.

[24] Verma A, Mishra A, Singh A & Agrawal A (2014) Effect of threshold voltage on various CMOS performance parameter. International Journal of Engineering Research and Applications 4(4): 21-28.

[25] Venkatachalam V & Franz M (2005) Power reduction techniques for microprocessor systems. ACM Computing Surveys 37(3): 195-237. DOI: http://dx.doi.org/10.1145/1108956.1108957.

[26] Subramanian K & Venkatachalam M (2018) Power reduction of shift register performance analysis and implementation of adaptive clock gating technology for low power systems. International Journal of Pure and Applied Mathematics 119(15): 357-366.

[27] Cadence Design Systems (2014) Power gating. URL: www.semiengineering.com/knowledge_centers/low-power/techniques/power-gating/. Accessed 4.2.2020.

[28] Chen Z, Wei L & Kaushik R (1997) Reducing glitching and leakage power in low voltage CMOS circuits using multiple threshold transistors. ECE Technical reports docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1084&context=ecetr.

[29] Haverinen J (2006) Lecture 5: Pipelining and Hazards. URL: www.ee.oulu.fi/research/tklab/courses/521480S/luennot/luento5.pdf. Accessed 15.6.2020.

[30] Jones N (2016) How to make embedded software smaller and faster. URL: www.barrgroup.com/embedded-systems/how-to/embedded-software-performance. Accessed 5.4.2020.

[31] Roy K & Johnson M. (1997) Software Design for Low Power. In: Nebel W & Mermet J (eds) Low Power Design in Deep Submicron Electronics. Dordrecht, Netherlands, Springer: 433-460.

[32] Batten D and D'Arcy P (1999) Intrinsic functions boost compilers. URL: www.eetimes.com/intrinsic-functions-boost-compilers. Accessed 25.3.2020.

[33] Acar H (2017) Software Development Methodology in a Green IT Environment. Doctoral thesis. University of Lyon, InfoMaths Doctoral School.

[34] Shah A & Kotecha K (2010) Efficient Scheduling Algorithms for Real-Time Distributed Systems. First International Conference on Parallel, Distributed and Grid Computing. Solan, India, 44-48. DOI: http://dx.doi.org/10.1109/PDGC.2010.5679608.

[35] Liu C & Layland J (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM 20(1): 46-61. DOI: http://dx.doi.org/10.1145/321738.321743.

[36] Davis R, Cucu-Grosjean L, Bertogna M & Burns A (2016) A review of priority assignment in real-time systems. Journal of Systems Architecture 65(C): 64-82. DOI: http://dx.doi.org/10.1016/j.sysarc.2016.04.002.

[37] Embedded (2002) Introduction to rate monotonic scheduling. URL: www.embedded.com/introduction-to-rate-monotonic-scheduling/. Accessed 15.6.2020.

[38] Chetto M (2012) Task scheduling in energy harvesting real-time embedded systems. Journal of Information Technology & Software Engineering 2(3): 1-2. DOI: http://dx.doi.org/10.4172/2165-7866.1000e106.

[39] Bambagini M, Hakan M, Aydin H & Buttazzo G (2016) Energy aware scheduling for real-time systems: A survey. ACM Transactions on Embedded Computing Systems 15(1): 1-34. DOI: http://dx.doi.org/10.1145/2808231.

[40] Yao F, Demers A & Shenker S (1995) A Scheduling Model for Reduced CPU Energy. Proceedings of IEEE 36th Annual Foundations of Computer Science.

Milwaukee, United States, 374-382. DOI: http://dx.doi.org/10.1109/SFCS. 1995.492493.

[41] Bini E, Butazzo G & Lipari G (2008) Minimizing CPU energy in real-time systems with discrete speed management. ACM Transactions on Embedded Computing Systems 8(4): 1-22. DOI: http://dx.doi.org/10.1145/1550987. 1550994.

[42] Pillai P & Shin K (2001) Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. Proceedings of the eighteenth ACM symposium on Operating systems principles. Banff, Canada, 89-102. DOI: http://dx.doi.org/10.1145/502034.502044.

[43] Pinheiro D, Concalves R, Valentin E, Oliveira H & Barreto R (2017) Inserting DVFS Code in Hard Real-Time System Tasks. Brazilian Symposium on Computer Systems Engineering. Curitiba, Brazil, 23-30. DOI: http://dx.doi.org/10.1109/SBESC.2017.10.

[44] Liu S, Qiu Q & Wu Q (2008) Energy Aware Dynamic Voltage and Frequency Selection for Real-Time Systems with Energy Harvesting. Design, automation and test in Europe. Munich, Germany, 236-241. DOI: http://dx.doi.org/10.1109/ DATE.2008.4484692.

[45] Islam F & Lin M (2017) Hybrid DVFS scheduling for real-time systems based on reinforcement learning. IEEE Systems Journal 11(2): 931-940. DOI: http://dx.doi.org/10.1109/JSYST.2015.2446205.

[46] Chen J, Wang S & Thiele L (2009) Proactive Speed Scheduling for Real-Time Tasks Under Thermal Constraints. IEEE Real-Time and Embedded Technology and Applications Symposium. San Francisco, United States, 141-150. DOI: http://dx.doi.org/10.1109/RTAS.2009.30.

[47] Luo G, Guo B, Shen Y, Liao H & Ren L (2009) Analysis and Optimization of Embedded Software Energy Consumption on the Source Code and Algorithm Level. International Conference on Embedded and Multimedia Computing. Jeju, South Korea, 1-5. DOI: http://dx.doi.org/10.1109/EM-COM.2009.5402965.

[48] Georgiou K, Xavier-de-Souza S & Eder K (2018) The IoT energy challenge: a software perspective. IEEE Embedded Systems Letters 10(3): 53-56. DOI: http://dx.doi.org/10.1109/LES.2017.2741419.

[49] Beningo J (2020) 5 RTOS best practices for designing RTOS-based applications. URL: www.beningo.com/5-best-practices-for-designing-rtos-based-applications. Accessed 6.2.2020.

[50] Texas Instruments (2018) Linux Core Power Management User's Guide (v4.4). URL: processors.wiki.ti.com/index.php/Linux_Core_Power_Management_User %27s_Guide_(v4.4). Accessed 3.2.2020.

[51] Walls C (2014) Power management considerations for efficient embedded systems development. URL: www.semiengineering.com/power-management-

considerations-for-efficient-embedded-systems-development/. Accessed 1.3.2020.

[52] Embedded (2002) Designing embedded software for lower power. URL: www.embedded.com/designing-embedded-software-for-lower-power/. Accessed 22.3.2020.

[53] Ripoll I & Ballaster-Ripoll R (2013) Period selection for minimal hyperperiod in periodic task systems. IEEE Transactions on Computers 62(9): 1813-1822. DOI: http://dx.doi.org/10.1109/TC.2012.243.

[54] Bertsekas D (2009) Convex Optimization Theory. Belmont, United States, Athena Scientific.

[55] Amazon Web Services Inc (2008) Trace hook macros. URL: www.freertos.org/rtos-trace-macros.html. Accessed 25.11.2019.

[56] Koskinen L, Hiienkari M, Mäkipää J & Turnquist M (2016) Implementing Minimum-Energy-Point Systems with Adaptive Logic. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24(4): 1247-1256. DOI: http://dx.doi.org/10.1109/TVLSI.2015.2442614.

[57] Hiienkari M, Gupta N, Teittinen J, Simonsson J, Turnquist M, Eriksson J, Anttila R, Myllynen O, Rämäkkö H, Mäkikyrö S & Koskinen L (2020) A 0.4-0.9V, 2.87pJ/Cycle Near-Threshold ARM Cortex-M3 CPU with in-Situ Monitoring and Adaptive-Logic Scan. IEEE Symposium on Low-Power and High-Speed Chips and Systems. Online, 556-558.

[58] Bini E & Buttazzo G (2001) A Hyperbolic Bound for the Rate Monotonic Algorithm. 13th Euromicro Conference on Real-Time Systems. Delfth, The Netherlands, 59-66. DOI: http://dx.doi.org/10.1109/EMRTS.2001.934000.

[59] Arm Limited (2016) URL: developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug. Accessed 3.12.2019.

[60] Segger Microcontroller GmbH (2019) J-Link / J-Trace User Guide. URL: www.segger.com/downloads/jlink/UM08001. Accessed 3.12.2019.

[61] Zhang Y, Suda N, Lai L & Chandra V (2017) Hello edge: keyword spotting on microcontrollers. arXiv:1711.07128v3 [cs.SD].

[62] Alawneh L & Abdelwahab H (2011) Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication. European conference of software maintenance and reengineering.

[63] Choi K, Soma R & Pedram M (2004) Dynamic Voltage and Frequency Scaling Based on Workload Decomposition. Proceedings of the 2004 International Symposium on Low Power Electronics and Design. 174-179. DOI: http://dx.doi.org/10.1109/LPE.2004.240891.

APPENDICES

Appendix 1. Software profiling and optimization implementation.

Preprocessing with a Python script removes any data from the trace file which is not relevant for profiling (Figure 26). The related events are extracted based on event abbreviations. Tasks, if necessary, can be removed from the trace if their task handle is known.

```
with open(in_file) as oldfile, open(out_file, 'w') as newfile:
    for line in oldfile:
        if (not any(delete_task in line for delete_task in delete_tasks)) and
        (any(event in line for event in event_abr)):
            if line_start in line:
                line = line.replace(line_start,"")
            newfile.write(line)
```

Figure 26. Code used for trace preprocessing.

The requirements for the trace file structure are set by the MATLAB implementation. The structure must match the implementation, or the input file cannot be correctly parsed. A processed trace file contains only event trace data (Figure 27). All trace prints have the same number of data fields which contain information which depends on the traced event. Data fields are separated by a colon. Additional data fields can be added to the end of the trace print as long as all traced events have the same number of data fields.

```
TC:Tmr Svc:20001724:647760532
CS-I:0005C930:12:647766539
CS-O:0005C930:CC:647788635
CS-I:00056010:12:647794028
CS-O:00056010:CC:647804688
CS-I:0005A4A0:12:647810061
CS-O:0005A4A0:CC:647816022
```

Figure 27. Example contents of a processed trace file.

Ideally, the trace file used for profiling contains a hyperperiod of the task set. Worst-case execution should be included. The largest execution cycle counts for tasks found in the trace are assumed to be the worst-case execution cycles of the tasks. Tracing is started from the beginning of the application execution to include task creation events.

Trace section is selected by hand. Automation of the trace section would require data mining. It is assumed that the user is aware of the task set properties which eases the section selection. When the section selection is done by hand, assumptions are made based on the optimization model implementation. The first event in the trace should be a task-switch-in event of an application task. This excludes the task-create events which are only used for task naming purposes and do not affect profiling. The last event in the trace section should be a switch-out event of the idle task. These assumptions cause that all tasks included in the trace are included as a whole and that there is possibility for frequency scaling even for the last executing application task in the trace.

Most of the data used in optimization is mined from the execution trace. The mined data will be displayed in dialog-boxes after profiling (Figure 28). All values can be changed. User-defined values must be defined at this stage. If a trace file is not used, user must provide all obligatory values for optimization. Detailed descriptions for all variables are given in Table 17 for system and runtime related parameters and Table 18 for task specific parameters.

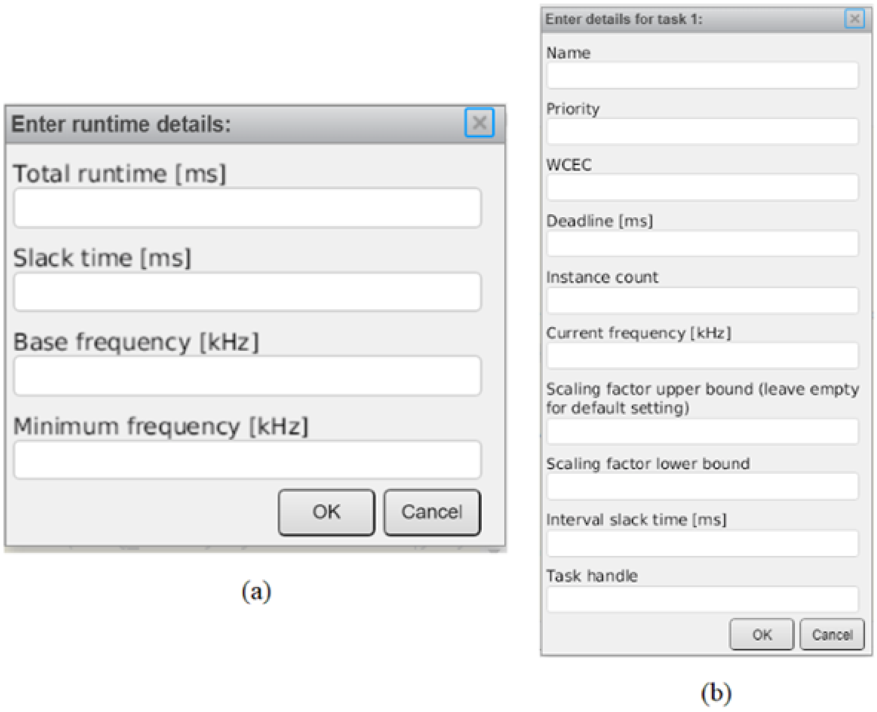


Figure 28. Dialog boxes which are used to define input values for optimization related to (a) system and runtime and (b) each task's execution.

Table 17. System and runtime parameter description

Parameter	Unit	Value from trace / value defined by user	Description
Total runtime	ms	Trace	Length of the trace which is used in optimization.
Slack time	ms	User	Time that is reserved for other activities than application tasks (included in the optimization model).
Base frequency	kHz	Trace, if frequency info in trace User otherwise	The execution frequency if frequency not changed during execution.
Minimum frequency	kHz	User	The minimum frequency is the lowest frequency level that can be used for execution. The value is (by default) used to define upper bound for scaling factor.

If a solution is found, optimization results are presented in the same order as the task-specific data was given in the dialog boxes i.e. first scaled frequency corresponds to task 1. Frequencies are defined according to Equation (4). In addition to scaled frequencies, the function which implements the whole optimization returns the output from the solver defined in [80], including the scaling factors. If a solution is not found, an error message specific to the solver is displayed.

Table 18. Task-specific parameter descriptions

Parameter	Unit	Value from trace / value defined by user	Description
Name	-	Trace	Descriptive name for a task. (optional)
Priority	-	User	Priority of a task (optional)
Handle	-	Trace	A pointer to the task control block. A task handle is used to identify a task. (optional)
WCEC	cycles	Trace	The largest amount of execution cycles defined from a task switch in to a switch out found in the trace.
Deadline	ms	User	The final target completion time defined from the start of a task
Instance count	-	Trace	The number of times a task has been switched in.
Current frequency	kHz	Trace, if frequency info in trace. User otherwise.	The execution frequency of a task. A single value for a task included in the optimization is assumed.
Scaling factor (UB)	-	User	Frequency scaling can be restricted by setting an upper bound for a scaling factor. (optional)
Scaling factor (LB)	-	User	Frequency scaling can be restricted by setting a lower bound for a scaling factor. (optional)
Interval slack time	ms	User	Interval slack time is reserved from the task time deadline value. It reduces the time available for frequency scaling of a task. This time is reserved for other activity than task execution.

Appendix 2. Optimization values for Ideal case of trace 1.

Optimization and parameters	
Variable name	Variable value
Volt change [ms]	0.376
Freq change [ms]	0.122
Cost cs in	volt_change + freq_change (0.4980)
CS overhead	0
Variable name	Variable value
Trace period [ms]	2321.0847
Guard time [ms]	0
Time available [ms] (difference of trace and guard time)	2321
Frequency [kHz] (for single frequency trace)	181 248
Minimum frequency [kHz]	12000

Variable name	Task names	
	Filter task	KWS task
Instance count	14	13
WCEC	839768	10089050
Relative deadline [ms]	100	100
Slack time [ms]	0	0
Exec. Freq. [kHz]	180 000	180 000
Actual freq. [kHz]	181 248	181 248

Appendix 3. Optimization values for Overhead-only case of trace 1.

Optimization and parameters		
Variable name	Variable value	
Volt change [ms]	0.376	
Freq change [ms]	0.122	
Cost cs in	volt_change + freq_change (0.4980)	
CS overhead (cycles)	2000	
Variable name	Variable value	
Trace period [ms]	2321	
Guard time [ms]	0	
Time available [ms] (difference of trace and guard time)	2321	
Frequency [kHz] (for single frequency trace)	181 248	
Minimum frequency [kHz]	12000	
Variable name	Task names	
	Filter task	KWS task
Instance count	14	13
WCEC	839768	10089050
Relative deadline [ms]	100	100
Slack time [ms]	0	0
Exec. Freq. [kHz]	180 000	180 000
Actual freq. [kHz]	181 248	181 248

Appendix 4. Optimization values for Restricted case of trace 1.

Optimization and parameters		
Variable name	Variable value	
Volt change [ms]	0.376	
Freq change [ms]	0.122	
Cost cs in	volt_change + freq_change (0.4980)	
CS overhead (cycles)	2000	
Variable name	Variable value	
Trace period [ms]	2321	
Guard time [ms]	200	
Time available [ms] (difference of trace and guard time)	2121	
Frequency [kHz] (for single frequency trace)	181 248	
Minimum frequency [kHz]	12000	
Variable name	Task names	
	Filter task	KWS task
Instance count	14	13
WCEC	839768	10089050
Relative deadline [ms]	100	100
Slack time [ms]	5	5
Exec. Freq. [kHz]	180 000	180 000
Actual freq. [kHz]	181 248	181 248

Appendix 5. Optimization values for trace 2.

Optimization and parameters		
Variable name	Variable value	
Volt change [ms]	0.376	
Freq change [ms]	0.122	
Cost cs in	volt_change + freq_change (0.4980)	
CS overhead (cycles)	2000	
Variable name	Variable value	
Trace period [ms]	34317.8574	
Guard time [ms]	0	
Time available [ms] (difference of trace and guard time)	34317.8574	
Frequency [kHz] (for single frequency trace)	181 248	
Minimum frequency [kHz]	12 000	
Variable name	Task names	
	Filter task	KWS task
Instance count	314	39
WCEC	839786	10089115
Relative deadline [ms]	100	100
Slack time [ms]	0	0
Exec. Freq. [kHz]	180 000	180 000
Actual Freq. [kHz]	181 248	181 248

Appendix 6. Optimization values for trace 3.

Optimization and parameters		
Variable name	Variable value	
Volt change [ms]	0.376	
Freq change [ms]	0.122	
Cost cs in	volt_change + freq_change (0.4980)	
CS overhead (cycles)	1000	
Variable name	Variable value	
Trace period [ms]	2320.365	
Guard time [ms]	0	
Time available [ms] (difference of trace and guard time)	2320.365	
Frequency [kHz] (for single frequency trace)	181 248	
Minimum frequency [kHz]	12 000	
Variable name	Task names	
	Filter task	KWS task
Instance count	14	13
WCEC	840403	10089638
Relative deadline [ms]	100	100
Slack time [ms]	0	0
Exec. Freq. [kHz]	180	180
Actual Freq. [kHz]	181248	181248

Appendix 7. Optimization values for trace 4.

Optimization and parameters	
Variable name	Variable value
Volt change [ms]	0.376
Freq change [ms]	0.122
Cost cs in	volt_change + freq_change (0.4980)
CS overhead (cycles)	1000
Variable name	Variable value
Trace period [ms]	908.7189
Guard time [ms]	0
Time available [ms] (difference of trace and guard time)	908.7189
Frequency [kHz] (for single frequency trace)	181 248
Minimum frequency [kHz]	12 000

	Task names	
Variable name	Filter task	KWS task
Instance count	4	7
WCEC	840202	10082251
Relative deadline [ms]	100	100
Slack time [ms]	0	0
Exec. Freq. [kHz]	180	180
Actual Freq. [kHz]	181248	181248