Techniques for dynamic hardware management of streaming media applications using a framework for system scenarios

Yahya H. Yassin[a,b], Francky Catthoor[c], Per Gunnar Kjeldsberg[a], Andrew Perkis[a]

[a] *Department of Electronic Systems*
Norwegian University of Science and Technology (NTNU),
Trondheim, Norway
[b] *Department of Electrical Engineering (ESAT)*
Katholieke Universiteit Leuven (KUL), Leuven, Belgium
[c] *IMEC, Kapeldreef 75, 3000 Leuven, Belgium, and*
part-time Professor at ESAT, KULeuven, Leuven, Belgium

Abstract

Many modern applications exhibit dynamic behavior, which can be exploited for reduced energy consumption. We employ a two-phase combined design-time/run-time methodology that identifies different run-time situations and clusters similar behaviors into system scenarios. This methodology is integrated with our framework for system scenario based designs, which dynamically tune the hardware to match the application behavior. We focus on streaming media applications, and achieve significant energy reductions for an extracted control structure of a video codec widely used in hand-held devices today. We encode a video stream consisting of different frame sizes based on measured available wireless bandwidth. Energy consumption is measured with a modified microcontroller board from Atmel with two alternative voltage and frequency settings. While maintaining the perceptual video quality and frame rate, our method results in up to 49% energy reduction for our encoded streams. The average tuning overhead of our mechanism is negligible after applying simple loop transformations to the encoder control structure. We furthermore show how to obtain up to 34.3% energy reductions for a system with limited slack available, by dynamically exploiting the available sleep modes.

Keywords: Dynamic Hardware Management, Energy Efficiency, System Architecture

1. Introduction

In embedded systems today, energy efficiency and power consumption is a major design constraint. Hand held devices are becoming more complex, because applications embedded in these devices require advanced functionality, in addition to increased parallelism. Furthermore, many high-end embedded systems today are dynamic in nature, and conventional embedded system design techniques are not able to meet these requirements in an energy efficiently way. State-of-the-art design methodologies try to cope with this

Email address: yassin@ntnu.no (Yahya H. Yassin)

challenge by identifying typical use-cases and dealing with them separately, in order to reduce the increased complexity of the system [1]. Conventional techniques focus mainly on static analysis where no dynamism is present. When dynamism is introduced, then only the worst-case behavior is used to perform the mapping. As an alternative, a well balanced combined design-time/run-time two-phase methodology, known as the system scenario based design [2], [3], has been developed. Instead of focusing on different use cases, this methodology exploits the application behavior at system level, resulting in a substantially larger optimization potential. This methodology consists of five design steps, and are described in Section 2.1. We use this methodology in our framework for system scenarios (FSS) [4].

Focusing on reducing the power consumption does not necessarily result in energy efficient designs. As an alternative, techniques such as race-to-halt (RTH) [5] can be used to speed up the execution of an application, thus increasing instantaneous power consumption, in order to finish early and achieve longer idle times. The static power consumed during idle times are reduced significantly due to the utilization of extremely low power states. RTH techniques thus reduces the total energy consumption of the system and is considered state of the art for CPU bound workloads when the arrival time of the next set of tasks to be processed is unknown. According to Awan et al. [5], Dynamic Voltage and Frequency Scaling (DVFS) techniques are on the other hand more suitable for memory bound workloads, and for real time systems with deadlines and long idle times.

In general, multimedia applications consume significant amounts of energy due to their high Quality of Service (QoS) requirements, as described by Hillestad [6]. Two-way interactive multimedia applications in particular have stringent requirements with respect to the quality of service delivered by the underlying transport network. Hillestad [6] generalizes two important requirements for IP-based streaming media applications. The first requirement is the real-time delivery to prevent buffer-underflow events. The second requirement is smooth rate changes to prevent oscillations in perceived quality (e.g., quality as measured subjectively by end users). For instance, if the frame size of a streaming video changes with less than 10 seconds intervals, or if the next frame size is significantly larger than the previous frame, then the perceptual video quality may affect the end-user's perception.

Interactive multimedia applications normally require the use of advanced video codecs, such as the widely used H264/AVC video codec [7]. Implementing the H264/AVC video codec in hand-held devices is a challenge by itself because of the requirements described by Hillestad [6] above, and because custom high performance digital signal processors (DSPs) are needed. These high performance DSPs are in general more power consuming than standard low-power of-the-shelf processors. On the other hand, general low power of-the-shelf processors struggle to process large video resolutions with acceptable processing times. Many researchers have therefore suggested multi-core solutions, where the H264/AVC codec is parallelized in order to concurrently process multiple frames or groups of pictures at the same time [8], [9].

Our goal is to not only improve the energy efficiency of the H264/AVC encoder, but also generally improve the energy efficiency for data-dependent applications with similar control structures as the H264/AVC encoder. Therefore, we focus on the control structure of the H264/AVC encoder, as obtained from the Mediabench II website [10], on which we apply our framework for system scenarios (FSS) [4]. In [11] our FSS framework is extended from only using Dynamic Frequency Scaling (DFS), to handling DVFS and RTH. The system scenarios approach is combined with RTH and DVFS on a SAM4L microcontroller board from Atmel [12], which contains an ARM Cortex M4 low power processor. The energy consumption is reduced by combining state-of-the-art techniques, i.e., DVFS and RTH, with the system scenario based design methodology [3] in our FSS framework. The experimentation is also substantially extended from our previous work [4]. In this journal extension of [11] we reduce the run-time overhead of our scenario mechanism with loop transformations resulting in negligible energy overhead. Our FSS framework is also further upgraded to select the best feasible sleep mode in the SAM4L board based on the remaining slack time available.

In Section 2.1 we give a brief overview of the system scenarios based design methodology presented by Gheorghita et al. [3], and in Section 2.2 we highlight the importance of targeting data-variables. Related work is presented in Section 3, and a motivational example is given in Section 4. Our experimental setup is explained in Section 5, how we identify our system scenarios is discussed in Section 6, and the implementation of our prediction and switching mechanism is presented in Section 7. We present and discuss the energy consumption in Section 8, and conclude our work in Section 9.

2. Terminology

2.1. The system scenario based design methodology

A two-phase design-time and run-time system scenario design methodology is presented by Gheorghita et al. [3]. It exploits the dynamic variation in an application, where system knobs, such as frequency, voltage, and low power modes, are tuned dynamically at run-time in order to gain significant reductions in energy consumption. The steps are summarized in Figure 1.

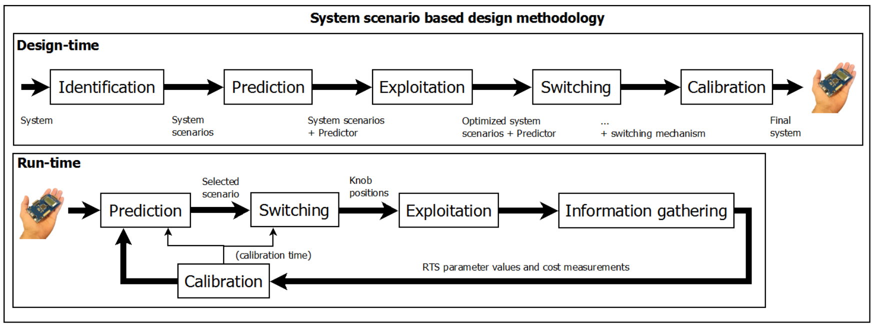


Figure 1: The system scenario based design methodology overview adapted from [3].

As shown in Figure 1, different run-time situations (RTSs) are identified at design-time, usually through profiling. RTSs with similar costs, e.g., execution time, energy consumption, or memory requirement, are clustered into scenarios. These system scenarios are different from typical use-case scenarios because they take detailed knowledge of the actual resource requirements into account, thus giving rise to more optimized designs [3].

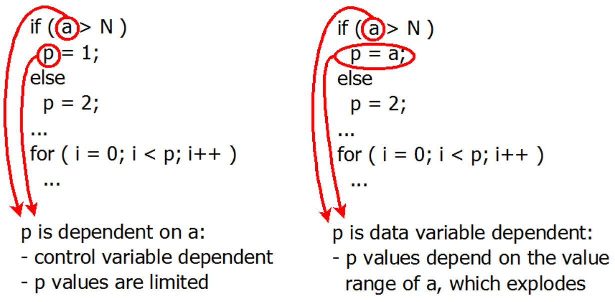
Based on the output from this identification step (Figure 1), a run-time scenario prediction mechanism is determined at design-time. At run-time, the prediction mechanism determines whether to switch from one system scenario to another based on dynamically changing system parameter values.

The exploitation step at design-time (in Figure 1) applies further optimizations to each scenario, similarly to what would have been done to the static implementation without the scenario approach, e.g., simple loop transformations. The system knob settings of each scenario are decided here. At run-time, the exploitation step is simply the execution of the scenario.

In order to change the system from one set of run-time platform configuration parameters (system knobs) to another, a scenario switching mechanism must be implemented. The switching mechanism to be used at run-time is determined at design-time. This mechanism will switch scenario based on the output of the run-time prediction step.

An optional calibration configuration can be designed (at design-time) and used (at run-time) in order to fine-tune the decision making.

2.2. Control variables vs data variables



```
if ( a > N )
  p = 1;
else
  p = 2;
...
for ( i = 0; i < p; i++ )
  ...
```

p is dependent on a:
- control variable dependent
- p values are limited

```
if ( a > N )
  p = a;
else
  p = 2;
...
for ( i = 0; i < p; i++ )
  ...
```

p is data variable dependent:
- p values depend on the value range of a, which explodes

Figure 2: Control vs data variable dependency

In order to better understand the challenge of data variable dependencies, Figure 2 illustrates a simplified code snippet illustrating the difference between a control variable and data variable dependency.

The code snippet on the left in Figure 2 illustrates a control variable dependency. The variable p in this case have a limited value range, and it will be relatively easy to monitor the application behavior if p is later used, for example, as a loop parameter.

The code snippet on the right shows a data variable dependent situation, where the value range of p equals the value range of a. If a is an input parameter, and p is used as a loop parameter in succeeding computations, then the loop size varies with the input parameter value range. Assuming the input parameter is a 16-bit integer value, then the value range of a will be equal to 2^{16}. This value range would simply produce too much run-time overhead for the scenario prediction and switching mechanisms.

3. Related work

Many researchers have optimized energy efficiency in embedded systems by proposing improved dynamic scheduling algorithms. Lee et al. [13] propose Leakage Control

Earliest-Deadline-First (LC-EDF) and Leakage Control Dual Priority (LC-DP) for reducing leakage current in hard real-time systems. The first scheduling algorithm schedules tasks according to their deadlines and the second scheduling algorithm schedules tasks with soft deadlines together with periodic, sporadic and adaptive tasks. Both algorithms focus on maximizing the available idle time of the system, and hence reducing the leakage by setting the system in sleep mode during idle times. Irani et al. [14] propose 3-competitive offline and constant competitive ratio online algorithms for power saving while considering shutdown in combination with DVFS. The offline algorithms considers the best energy efficiency of all available power modes for a given task. The online algorithm uses the EDF algorithm to schedule arriving jobs in a way that minimizes total energy and meet their deadlines. Linwei et al. [15] uses a combination of DVFS and shutdown to minimize the overall energy consumption based on the latest arrival time of jobs. In a hard real-time system scheduled by the EDF strategy they reduce the energy consumption during idle intervals by utilizing DVFS to increase the speed of a task. Most of these approaches produce significant run-time analysis overhead. Awan et al. [5], propose a solution with significantly less run-time overhead. They combine an online and offline algorithm that selects the best feasible sleep mode based on the available time slack for a given workload. All these techniques have mainly one thing in common; scheduling of different tasks in combination with RTH or DVFS. Other researchers focus on machine-learning techniques to predict degradation in performance [16]. Such techniques are complex to implement on embedded platforms, and focus mainly on use cases with shared resources in multicore platforms.

A method for automatic identification of system scenarios that incorporate correlations between different parts of applications is proposed by Gheorghita et al. [17], [18]. When many-valued parameters are considered, the overhead for scenario prediction is significant, however. Hammari et al. [19] propose a method based on the projection of scenarios onto an RTS parameter space, where many-valued parameters are handled with the use of polyhedral techniques. These techniques are needed to fully automate the scenario analysis, but is not required for partly automated systems where the scenario information is inserted by the designer.

In general, the scenario prediction mechanism is highly dependent on how the identification step is performed, as indicated by previous work [17], [18], [19]. The authors in previous work conclude that the number of variables used in the prediction should be low. This is also the same for other identification approaches, e.g., the method of using loop-trip count profiling [20]. Previous work mainly focuses on control structures in applications with limited or no data variable dependency. When data dependent RTS-parameters dominates the application behavior, the value range increases significantly. Increased value range in the RTS parameter space usually increases the run-time prediction overhead.

Damavandpeyma et al. [21] use DVFS as a system knob, and propose a compact scenario aware data flow graph (SADF). Their approach assumes that the system scenarios have been clustered as part of the identification step, where the clustered result does not contain any many-valued parameters, and the number of control- and/or data-variables used in their analysis is limited. Their approach works well for control variables with limited dependency variation. Another approach is needed for data variable dependency where the data variables have a wide range of possible values.

The data variable challenge increases with the combination of multiple data-dependent

RTS parameters and nested loops. Compared with the previous work described above we here combine the FSS framework from our previous work [4] with DVFS and RTH online within a running task to achieve significant energy reductions when a data variable dependency is present. The data variable dependency is handled by using inline functions that combine multiple RTS parameters, and classify different ranges as scenarios. Low overhead and globally placed detection equations are defined that can detect the required cycle count of the application. The output of these detection equations are input to our prediction mechanism. The prediction mechanism then compares its input to different ranges in a lookup table, before deciding which scenario to choose. Our approach is scalable with increasing numbers of system knobs and an increased number of RTS-parameters within an application. In the following sections we will show how we were able to achieve significant energy reductions with our FSS framework implemented on a SAM4L microcontroller board.

4. Motivational example

Rapid advances in processor and wireless networking technology are opening up for a new class of streaming media applications for mobile hand-held devices such as mobile phones and tablets [22]. According to Mohapatra et al. [22], these devices have inadequate resources, such as lower processing power, memory, display capabilities, storage and limited battery lifetime as compared to desktop and laptop systems. Video communication via the internet on such devices have become more common in recent years through the use of programs such as Skype, Viber, and Hangouts. A major challenge with hand-held devices is that the power consumption increases significantly during a video call. The phone's camera is used to encode video and stream it over the internet continuously. At the same time, video from the caller is received and decoded continuously. Considering that video is streamed with a rate of 24-30 frames per second or more, the amount of simultaneous encoding and decoding is significant. One of the commonly known challenges is that the user perception of a video call can vary based on the internet connection speed, the processing limitations of the hand-held device, and battery level. An advanced codec such as H264/AVC is suitable for hand-held devices [7], and is widely deployed today. The H264/AVC codec is far more advanced than its predecessors, and introduce more dynamism, especially with varying input frame sizes. This dynamicity can be exploited using the system scenarios design methodology presented by Gheorghita et al. [3].

There exist proposals for video resizers today, such as the "Apparatus and method for resizing an image" [23], which rescales a camera's video frame size. A simple control logic could easily be added to change the frame size of a picture, such that the dynamic behavior of the internet connectivity could be exploited and used to decide the scale factor of the video before it is encoded and transmitted.

Figure 3 shows the results of measurements we have performed of available internet bandwidth for a 500-second period on three different wireless networks. The upper graph shows the wireless local area network (WIFI) at NTNU, while the middle and lower graphs shows measurements over mobile Long-Term Evolution (LTE) [24] and Wideband Code-Division Multiple-Access (WCDMA) [25] networks, respectively. The measurements are performed using the bandwidth measurement tool IPERF [26]. IPERF is used to measure available bandwidth between two IP-addresses. Our measurements are performed with 10 seconds intervals between a Sony Xperia Z3 android-based smartphone and a computer running Ubuntu. Figure 3 clearly shows significant dynamicity in the available bandwidth, which we can exploit by using our FSS-framework [4]. In addition we calculate the moving average over the available bandwidth, as seen in Figure 3.

We focus on the H264/AVC encoder control structure obtained from the Mediabench II website [10], where we adapt the platform's voltage and frequency according to the input frame size, assuming that a video resizer circuit rescales the input frames based on the moving average measurement of the available bandwidth in Figure 3.

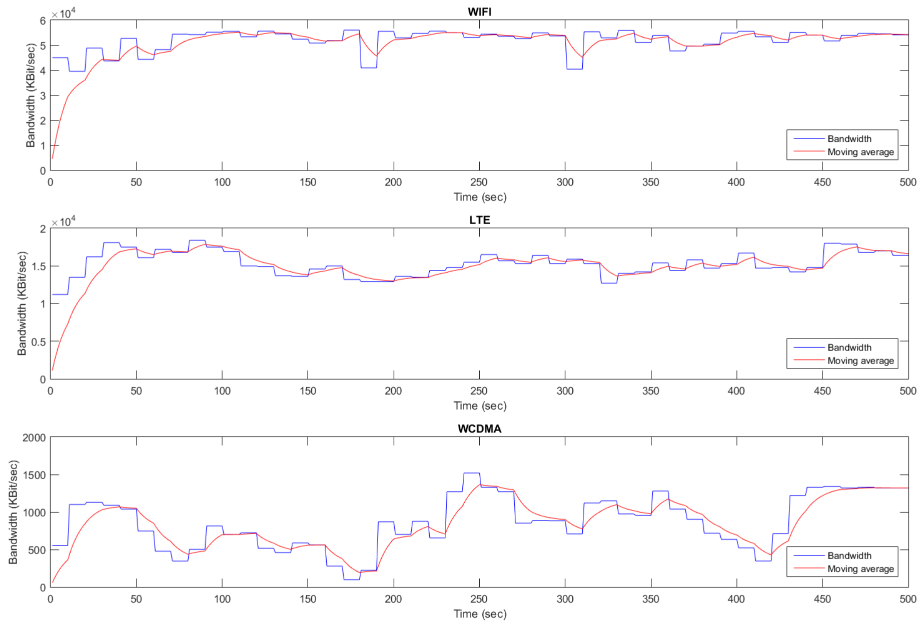


Figure 3: Available bandwidth measurement and moving average over WIFI, LTE and WCDMA

5. Experimental setup

Our target domain is embedded systems such as mobile phone platforms where energy efficiency is the main constraint. Flora et al. [27] highlight the main hardware characteristics of such a system, containing relatively weak CPUs and limited memory. Their survey show that good architectures are even more important for mobile phones than for desktops. The SAM4L board from Atmel [12] is a representative example of a mobile platform because it contains an energy friendly CPU with limited on-chip SRAM. One advantage of the SAM4L-board is the application control of active voltage states, making it possible to dynamically tune the hardware for the running application.

During our experiments we measure the live current consumption and core voltage on the SAM4L board using a NI myDAQ measurement device [28]. The processor core voltage is measured directly on the vcc_mcu_core pin of the SAM4L microcontroller relative to the board ground, while the SAM4L microcontroller's total current consumption is measured through available measurement pins on the board. Time stamped data is logged with a locally developed LabVIEW program, and used together with our voltage and current measurements, to find the energy consumption of the H264/AVC encoder control structure. The single-core FSS framework is used to implement the H264/AVC encoder control structure with system scenarios.

Our version of the SAM4L board contains the ATSAM4LC4CA microcontroller. The microcontroller contains an ARM Cortex M4 processor, and supports two different run modes. RUN0 operates with a core voltage of 1.8V, while RUN1 operates at 1.2V. The microcontroller also supports various low power states, i.e., sleep modes, where the processor clock is turned off, and multiple oscillators with different frequency ranges. Table 1 shows all sleep modes used in our experiments and their corresponding measured power consumption, overhead and wakeup times.

Table 1: Run/Sleep modes in our SAM4L-board from Atmel

Run/Sleep mode	Measured power consumption (μW)	Wakeup time (incl. setup time)
RUN0 (heavy load)	29 865	N/A
RUN0 (idle)	25 340	N/A
sleep00	14 010	83 clk @ 40 MHz
sleep01	10 370	659 clk @ 40 MHz
sleep02	5 460	664 clk @ 40 MHz
sleep03	79	3907 clk @ 40 MHz
wait0	12	5669 clk @ 40 MHz
retention0	5.8	5997 clk @ 40 MHz
RUN1 (heavy load)	5 107	N/A
RUN1 (idle)	4 013	N/A
sleep10	2 440	88 clk @ 12 MHz
sleep11	1 830	390 clk @ 12 MHz
sleep12	1 460	395 clk @ 12 MHz
sleep13	51	1057 clk @ 12 MHz
wait1	5.9	1728 clk @ 12 MHz
retention1	4.4	6703 clk @ 40 MHz 1786 clk @ 12 MHz

For the scope of our research, we have selected the internal oscillators RCFAST (12 MHz) and RC80M (80 MHz) as the main clock sources for our DVFS settings. Since the ARM Cortex M4 processor only supports frequencies up to 48 MHz, the RC80M oscillator is clock divided to 40 MHz when it is selected as the main clock source. Hence, our DVFS settings are PS0 (1.8 V and 40 MHz) and PS1 (1.2 V and 12 MHz).

Based on the SAM4L board datasheet, the DVFS is triggered by a write to configuration registers after a write to protective lock registers. In other words, a switch is triggered after a few clock cycles. We have measured the switching time between our two clock settings with an oscilloscope. The switch takes approximately 15 clock cycles (at 12 MHz) when both oscillators are on continuously, and approximately 50 clock cycles (at 12 MHz) if the currently unused oscillator is powered down. The added power consumption by having both oscillators enabled is negligible because unused oscillators that are idling does not contribute to any measurable power consumption. Therefore, both oscillators are kept enabled in our measurements for a significantly reduced switching time.

We have measured the total static power to be less than 16% of the total power consumption in PS0, and less than 29% in PS1. The static power consumption on the SAM4L board is approximated by measuring the total power consumption for two different frequencies in each power mode. With these four equations we estimate the effective capacitance and static energies under the assumption that the activity factor (α) is equal to 0.5 in Equation 1.

$$P_{\text{Total}} = P_{\text{Static}} + P_{\text{Dynamic}} = P_{\text{Static}} + \alpha C V^2 f \tag{1}$$

When performing dynamic voltage scaling on the SAM4L board, the main clock frequency supported in the new voltage level has to be configured first. According to the datasheet, the 1.2 V (PS1) configuration does not support frequencies larger than 12 MHz. The frequency setting must therefore be switched from 40 MHz to 12 MHz before the voltage switch from 1.8 V (PS0) to 1.2 V (PS1). When changing configuration from PS1 to PS0 the frequency must be scaled after the voltage switch to maintain stability.

In addition, the output from the voltage regulator has to be stable after a voltage switch before the CPU can continue processing. After a switch from PS1 to PS0 we have to wait for a status flag, which signals that the voltage level has stabilized. The CPU is therefore stalled while waiting for this flag to be set in the SAM4L board. However, when a voltage switch is triggered from PS0 to PS1 then waiting for this flag is not necessary before continuing the CPU execution. As described by Atmel Support, when switching from a higher voltage level to a lower level, the voltage regulator is stable and will regulate fast enough to continue execution in succeeding cycles. The measured current consumption will gradually decrease after this switch until the charge on the decoupling capacitor connected to the core voltage has been consumed down to its PS1 level. The power delivered from the regulator on the other hand is equal to the PS1 level immediately after the voltage switch.

Our SAM4L board came initially with a 22 µF decoupling capacitor connected between the core voltage input and ground. The large decoupling capacitor resulted in extremely long switching times. This was confirmed by Atmel Support, and they recommended a change to 4.7 µF to guarantee that all modules would function properly. Since we are only interested in the CPU, we measured the DVFS switching times with lower decoupling capacitors. Our experiments showed that the CPU functioned correctly using a 1 µF decoupling capacitor, giving a DVFS switching time comparable to state-of-the-art processors. Table 2 shows the switching times and energy overhead when switching from PS0 to PS1 and vice versa. The energy overhead is found by measuring the time between the last instruction before and the first instruction after the DVFS switch, before multiplying this time with the power consumption for the run mode before the switch. An oscilloscope was used for the timing measurement.

6. Scenario identification methodology

The application behavior of streaming media applications must be profiled with different input data in order to detect different run-time situations (RTSs) that occur while

Table 2: DVFS switching times between PS0 and PS1.

From	To	Cycles @ 12 MHz	Switching time (μS)	Overhead Energy (μJ)
PS1	PS0	197	16.40	0.0623
PS0	PS1	15	1.25	0.0281

the application is running. Our scenario identification approach takes into account available and relevant application domain parameters. This is typically the stream of input data, the application code handling this data, long input sequence which we profile in order to identify the dynamism in the workload, application run time due to while loops/conditions, data variable dependency, and storage requirements.

The H264/AVC encoder source code was obtained from the Mediabench II website [10]. This model is profiled at design-time in order to explore the dynamic range of loop iteration count variations for various input frame sizes. The H264/AVC encoder control structure is extracted from the Mediabench II source code, and modeled under the assumption that all memory accesses take one clock cycle. In [11] all instructions, i.e., calculations and memory accesses, are exchanged with ”no operation” (NOP) instructions, which makes the model highly optimistic. In the current work we exchange these NOPs with single cycle memory access to SRAM in order to make the model more realistic. These improvements are discussed in further detail in Section 7.2.

In addition, different voltage and frequency settings are profiled in order to find optimal DVFS configurations in combination with RTH. Our profiled results show that the optimal energy consumption is achieved if the maximum available frequency setting for each voltage level is selected. This results in two scenarios, PS0 (1.8V and 40 MHz) and PS1 (1.2V and 12 MHz).

Table 3 shows an evenly distributed selection of true 16:9 resolutions that we have selected, which our SAM4L board can process within a one second time frame using our extracted model. It also shows the corresponding scenario selections and calculated run times, based on our available power modes in the SAM4L board. In Table 3, the Scenario* column represents the scenario distribution for a platform with more available voltage settings, and is discussed in Section 8.

The largest frame size the platform can handle from our selection is measured to be 816 x 459. The required run time (in clock cycles) for each frame size is estimated by counting all loop iterations and multiplying with all instructions within each loop based on the program dis-assembly of the extracted model. These numbers are compared to the measured run time for each frame size. From the measured results, we observed that the correct cycle count could be approximated by multiplying the number of all loop iterations for any input frame size with a factor of 4.22. This factor corresponds to the additional assembly instructions needed to determine the loop length and initiate for-loops before they are executed. The number of all loop iterations is calculated as shown in Listing 1. The total number of iterations are combined together with other parameters for all the nested loops in the application, resulting in the loopcnt variable. This loopcnt variable is multiplied by the approximated factor of 4.22. The separation of the approximated factor allows for easy portability of the scenario prediction mechanism to other platforms, because other platforms might implement the loop handling instructions differently. This difference would result in a higher or lower factor. The largest frame processing time is therefore approximated to 34.7 million clock cycles in an un-optimized setting where no pipelines are utilized. We use this frame size processing time as our worst-case scenario, and we exploit it in combination with DVFS and RTH for lower frame sizes. We benefit from DVFS if the processing time of a smaller frame size with a lower frequency setting is less than the processing time of our worst-case frame size. Therefore, we have to take our available system knobs into consideration when we classify our system scenarios.

Table 3: Scenario assignment based on run time

Input frame size	Run time (million cycles)	Scenario	Scenario*
816 x 459	34.70	0	0
752 x 423	29.54	0	0
688 x 387	24.79	0	1
624 x 351	20.46	0	2
560 x 315	16.54	0	2
496 x 279	13.04	0	3
432 x 243	9.96	1	4
368 x 207	7.29	1	4
288 x 162	4.54	1	5
224 x 126	2.80	1	5

Our scenario selection will be dependent on the platform's running frequency. In PS1 the platform is processing a frame 70% slower than in PS0. Therefore, frame sizes that need more than 30% of the run time of our worst-case frame size are processed with PS0 (Scenario 0), and smaller frame sizes are processed with the PS1 configuration (Scenario 1).

Our arguments above are generalized in Equation 2, where we set the Scenario 1 upper limit for the maximum frame processing time in PS1 to be equal to the frame processing time of our largest frame (816 x 459) in PS0. The Scenario 1 upper limit is calculated as shown in Equation 3. In Equation 2 and 3, $T_{\text{WC frame}}$ is the execution time in seconds of the worst-case frame size. T_{sc_1} is the maximum allowed execution time for Scenario 1 in seconds. LC_{sc_0} and LC_{sc_1} are the total number of loop iterations for a specific frame in PS0 or PS1, respectively. β is the approximated scale factor used to get the total number of clock cycles, and f_{sc_0} is the platform's main clock frequency when Scenario 0 is active. In other words, if the required frame processing time is above 30% of our worst-case frame processing time, then the platform switches from Scenario 1 to 0. Based on Equation 2, we find the Scenario 1 limit in Equation 3 without involving the β scale factor. This simplification makes our method portable to other platforms with a different β scale factor. Equation 3 shows the limit 2,466,807, which corresponds to 10.4 million clock cycles of actual run time (after multiplying with the 4.22 approximated scale factor β). I.e., all frame sizes with a run time above 10.4 million clock cycles are assigned to Scenario 0 in Table 3.

$$T_{\mathrm{WC\ frame}} = T_{\mathrm{sc}_1} = \frac{\beta \cdot LC_{\mathrm{sc}_0}}{f_{\mathrm{sc}_0}} = \frac{\beta \cdot LC_{\mathrm{sc}_1}}{0.3 \cdot f_{\mathrm{sc}_0}} \tag{2}$$

$$LC_{\mathrm{sc}_1} = 0.3 \cdot LC_{\mathrm{sc}_0} = 0.3 \cdot 8,222,691 = 2,466,807 \tag{3}$$

7. Prediction and switching mechanism

Our generic framework for system scenarios (FSS) [4] is used to implement the scenario mechanisms as shown, e.g., in the left hand side of Figure 5 on page 15.

The FSS framework separates the application software from the scenario mechanisms to make the framework portable to other platforms. This separation allows us to combine software and hardware implementations of the system scenario mechanisms with minimal modifications to the original application. I.e., the application monitoring parts of the scenario mechanism is defined by the Application Monitoring Unit (AMU). Lightweight and inline AMU functions are injected in the application at different checkpoints to collect RTS information, which will be used to predict or detect future system scenarios. The AMU then calls the Platform Adaption Manager (PAM), which performs the actual platform reconfiguration for the detected or predicted scenario.

The AMU is separated from the PAM because the PAM contains functions and non-portable configurations which are specific to the platform. If the FSS framework is ported to another platform, then the PAM functions and configurations must be configured for the new platform as well. The AMU functions would in this case stay the same and is platform independent. Alternatively, the AMU could be implemented partly in software and partly in hardware to speed up the scenario prediction, which may be needed in multi-core systems using system scenarios if the communication between off-chip modules takes too long time with a software-only approach. The hardware implementations of our FSS framework is out of the scope of this paper.

All streaming media applications where the application run time is dependent on the input frame size would benefit from this approach because the nested loop structures in such applications normally have a strong dependency on the frame size, i.e., larger frame sizes results in significantly increasing the storage requirements and the application run time.

The input data variable dependency of H264/AVC encoder control structure is modeled as part of our application monitoring unit (AMU) in the FSS framework [4]. The input frame sizes of the H264/AVC encoder is used directly in the calculations to determine succeeding loop sizes and memory accesses. This dependency and how we exploit these calculations is discussed in Section 7.1. In Section 7.2 and 7.3 we present two new contributions to this paper, where our AMU mechanisms are improved by the use of loop transformations (Section 7.2), and a sleep mode management is implemented to select the lowest power consuming sleep mode allowed from the calculated slack time after a processed frame (Section 7.3).

7.1. Input data variable dependency prediction

The input data variable dependency is shown in Listing 1, where inline application dependent equations are pre-calculated (lines 2-7), before a global lookup-table is updated with the combined RTS parameter (line 10) as part of the overall AMU. This RTS parameter is then read by the AMU_1 function, which detects the next scenario (line 13). The application dependent calculations in lines 2-6 correspond to succeeding (nested) for- and/or while-loops in the encoder control structure, which are combined together in line 7 to estimate the required frame processing time in terms of loop iterations. In our particular case it would have been enough to compare the loopcnt variable with the threshold between Scenario 0 and 1 directly, and use the result to determine the next scenario.

However, we have generalized this comparison by using the AMU_1 function. This lightweight inline function compares the global RTS lookup-table values with global scenario lookup table values in order to determine which scenario the platform will adapt to. This generalization takes into account multiple scenario parameters and RTS parameters. The use of lookup tables would simplify an extension to multiple independent RTS parameters and/or scenarios. The values in the global scenario lookup table is found from our identification step, from Equation 3. If the value of the global RTS lookup table is greater than the value in the global scenario lookup table, then the platform changes its configurations to PS0 if it is in PS1, and vice versa. The PAM function is then called by the AMU and performs the platform re-configuration using the DVFS system knob as described in Section 5. Figure 4 shows the rts_data content for different input frame sizes which corresponds to Table 3 after multiplying the values with the β-factor.

```
1   // Functions for monitoring RTS parameters
2   long loopcnt1 = (size_y + img_pad_size + img_pad_size) * (2 + (3*(size_x +
        img_pad_size + img_pad_size)));
3   long loopcnt2 = ((size_x + 2 * img_pad_size) * 2) * (5 + ((3*(maxy - 2 - 2)) +
        (3*(ypadded_size - maxy + 2))));
4   long loopcnt3 = ((je2 + 2)/2) * (2 + (((ie2 - 2)/2) + 2));
5   long loopcnt4 = (ie2 + 2) * (((je2 - 2)/2) + 2);
6   long loopcnt5 = (size_y/4) * (1 + (6*(size_x/4)));
7   long loopcnt = loopcnt1 + loopcnt2 + loopcnt3 + loopcnt4 + loopcnt5;
8
9   // Update combined RTS parameter
10  (g_rts_data)[0] = loopcnt;
11
12  // Scenario detection/prediction
13  AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]);
```

Listing 1: Functions combining data-dependent RTS parameters

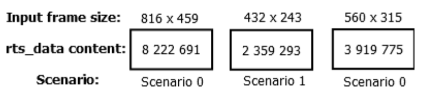


Figure 4: Contents of rts_data for different input frame sizes.

7.2. Improving the scenario energy overhead

Our initial approach as described in Section 7.1 resulted in an average energy overhead of 8.5% for the 500-second sequence [11]. These measurements were taken using an extreme conservative approach, however, where the scenario mechanism is triggered between each frame even if the scenario for the next frame is the same as the previous one. In this paper, we have improved our mechanism to trigger a DVFS switch only if the previous scenario is different from the predicted scenario. This improvement reduced the average energy overhead to 7.3%.

Further investigations to our encoder control structure reveled that our NOP instructions only delays further processing on the SAM4L board. Therefore, the measured power consumption of our encoded frames were equal to the boards idle power consumption with no read and/or writes to memory, which is very optimistic. We therefore exchanged the NOPs in our encoder control structure with one cycle read and write operations to the board's on-chip SRAM. This modification resulted in more realistic power measurements for video encoding compared to only using NOPs. We measured the energy consumption for our 500-second streams with the new encoder control structure model, which resulted in a 2.4% energy overhead using our scenario mechanisms. This overhead corresponds to the calculations performed in Listing 1.

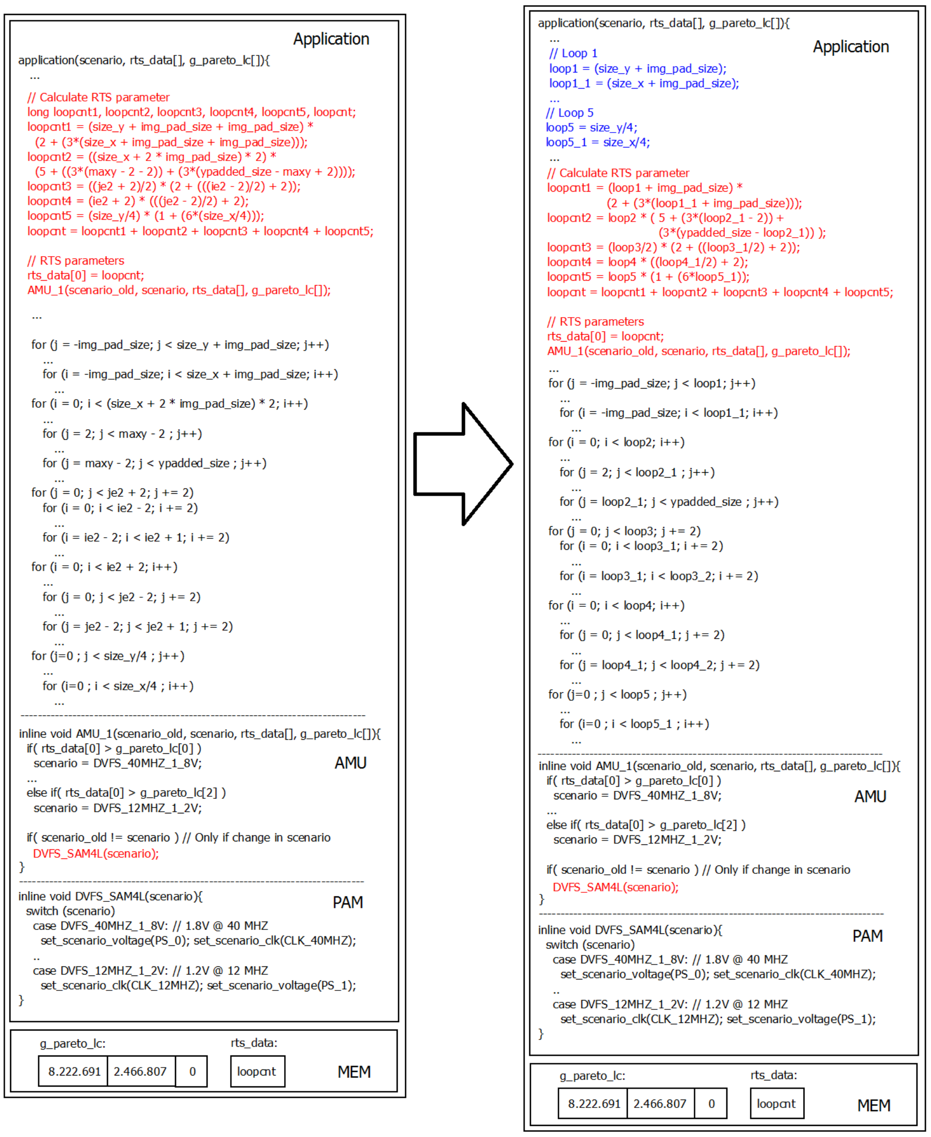
Finally, we observed that parts of these calculations are performed before each for-loop in the code, making our pre-calculations in Listing 1 redundant. Therefore, simple loop transformations are applied, where the loop size calculation performed before each for-loop is performed before executing the encoder control structure. As a result, our scenario mechanisms could benefit from this improvement and eliminate redundant calculations. The average energy overhead of our scenario mechanisms are measured to be 0.7% after the simple loop transformations, which we consider negligible. Figure 5 illustrates the changes after the simple loop transformations.

7.3. Introducing sleep mode management with system scenarios

Our SAM4L board contains different sleep modes as shown in Table 1 on page 8. Other work, such as the RTH energy saving strategy by Awan et al. [5], have exploited available sleep modes to utilize RTH more energy efficiently. Their solution is based on an offline and online approach, where they find the most efficient sleep modes offline for a set of jobs based on the maximum time interval, t_1, for which the processor may be enforced in a sleep state without causing any task to miss its deadline under worst-case assumptions. Their online algorithms compute the sleep time by calculating the available slack present at run-time. The most efficient sleep state is then selected online based on the sleep time calculated online and the pre-calculated t_1 interval. Their approach works well for sporadic tasks where jobs can start at any time with negligible online overhead.

Our system scenario approach enables us to simplify the solution by Awan et al. significantly by exploiting our pre-calculated loop execution times from our AMU_1 mechanism. Instead of computing the slack online, we pre-compute the total available frame cycles (TOTAL_FRAME_CYCLES in Listing 2) offline for each scenario (PS0 and PS1) based on what we know about our system from the scenario identification step. The stored wakeup time, including the setup time, are then summed with the calculated loopcnt value (*g_rth_lc* and *g_rts_data* in Listing 2) and compared to the pre-calculated total frame cycles available online. Our solution then selects the lowest power consuming sleep mode if our estimated total execution time is less than the slack time available. The different sleep power consumptions, setup times and wakeup times are measured on our SAM4L board, and included in Table 1. All these wakeup time values (including the setup time) from Table 1 are stored in the *g_rth_lc* global lookup table. This lookup table is read by the AMU_2 mechanism shown in Listing 2.

AMU_2 reduces the online overhead complexity from Awan et al. to an if-else comparison. Starting with the lowest power consuming sleep mode, the AMU_2 compares



```
application(scenario, rts_data[], g_pareto_lc[]){
  ...
  // Calculate RTS parameter
  long loopcnt1, loopcnt2, loopcnt3, loopcnt4, loopcnt5, loopcnt;
  loopcnt1 = (size_y + img_pad_size + img_pad_size) *
    (2 + (3*(size_x + img_pad_size + img_pad_size)));
  loopcnt2 = ((size_x + 2 * img_pad_size) * 2) *
    (5 + ((3*(maxy - 2 - 2)) + (3*(ypadded_size - maxy + 2))));
  loopcnt3 = ((je2 + 2)/2) * (2 + (((ie2 - 2)/2) + 2));
  loopcnt4 = (ie2 + 2) * (((je2 - 2)/2) + 2);
  loopcnt5 = (size_y/4) * (1 + (6*(size_x/4)));
  loopcnt = loopcnt1 + loopcnt2 + loopcnt3 + loopcnt4 + loopcnt5;

  // RTS parameters
  rts_data[0] = loopcnt;
  AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]);

   ...

   for (j = -img_pad_size; j < size_y + img_pad_size; j++)
     ...
     for (i = -img_pad_size; i < size_x + img_pad_size; i++)
       ...
   for (i = 0; i < (size_x + 2 * img_pad_size) * 2; i++)
     ...
     for (j = 2; j < maxy - 2 ; j++)
       ...
     for (j = maxy - 2; j < ypadded_size ; j++)
       ...
   for (j = 0; j < je2 + 2; j += 2)
     for (i = 0; i < ie2 - 2; i += 2)
       ...
     for (i = ie2 - 2; i < ie2 + 1; i += 2)
       ...
   for (i = 0; i < ie2 + 2; i++)
     ...
     for (j = 0; j < je2 - 2; j += 2)
       ...
     for (j = je2 - 2; j < je2 + 1; j += 2)
       ...
   for (j=0 ; j < size_y/4 ; j++)
     ...
     for (i=0 ; i < size_x/4 ; i++)
       ...
-------------------------------------------------------------------------------
inline void AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]){
  if( rts_data[0] > g_pareto_lc[0] )
   scenario = DVFS_40MHZ_1_8V;
  ...
  else if( rts_data[0] > g_pareto_lc[2] )
   scenario = DVFS_12MHZ_1_2V;

  if( scenario_old != scenario ) // Only if change in scenario
   DVFS_SAM4L(scenario);
}
-------------------------------------------------------------------------------
inline void DVFS_SAM4L(scenario){
  switch (scenario)
   case DVFS_40MHZ_1_8V: // 1.8V @ 40 MHZ
    set_scenario_voltage(PS_0); set_scenario_clk(CLK_40MHZ);
   ..
   case DVFS_12MHZ_1_2V: // 1.2V @ 12 MHZ
    set_scenario_clk(CLK_12MHZ); set_scenario_voltage(PS_1);
}
```

```
application(scenario, rts_data[], g_pareto_lc[]){
  ...
  // Loop 1
  loop1 = (size_y + img_pad_size);
  loop1_1 = (size_x + img_pad_size);
  ...
 // Loop 5
 loop5 = size_y/4;
 loop5_1 = size_x/4;
  ...
 // Calculate RTS parameter
 loopcnt1 = (loop1 + img_pad_size) *
                (2 + (3*(loop1_1 + img_pad_size)));
 loopcnt2 = loop2 * ( 5 + (3*(loop2_1 - 2)) +
                         (3*(ypadded_size - loop2_1)) );
 loopcnt3 = (loop3/2) * (2 + ((loop3_1/2) + 2));
 loopcnt4 = loop4 * ((loop4_1/2) + 2);
 loopcnt5 = loop5 * (1 + (6*loop5_1));
 loopcnt = loopcnt1 + loopcnt2 + loopcnt3 + loopcnt4 + loopcnt5;

 // RTS parameters
 rts_data[0] = loopcnt;
 AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]);
  ...
 for (j = -img_pad_size; j < loop1; j++)
   ...
   for (i = -img_pad_size; i < loop1_1; i++)
     ...
 for (i = 0; i < loop2; i++)
   ...
   for (j = 2; j < loop2_1 ; j++)
     ...
   for (j = loop2_1; j < ypadded_size ; j++)
     ...
 for (j = 0; j < loop3; j += 2)
   for (i = 0; i < loop3_1; i += 2)
     ...
   for (i = loop3_1; i < loop3_2; i += 2)
     ...
 for (i = 0; i < loop4; i++)
   ...
   for (j = 0; j < loop4_1; j += 2)
     ...
   for (j = loop4_1; j < loop4_2; j += 2)
     ...
 for (j=0 ; j < loop5 ; j++)
   ...
   for (i=0 ; i < loop5_1 ; i++)
     ...
-------------------------------------------------------------------------------
inline void AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]){
  if( rts_data[0] > g_pareto_lc[0] )
   scenario = DVFS_40MHZ_1_8V;
  ...
  else if( rts_data[0] > g_pareto_lc[2] )
   scenario = DVFS_12MHZ_1_2V;

  if( scenario_old != scenario ) // Only if change in scenario
   DVFS_SAM4L(scenario);
}
-------------------------------------------------------------------------------
inline void DVFS_SAM4L(scenario){
  switch (scenario)
   case DVFS_40MHZ_1_8V: // 1.8V @ 40 MHZ
    set_scenario_voltage(PS_0); set_scenario_clk(CLK_40MHZ);
   ..
   case DVFS_12MHZ_1_2V: // 1.2V @ 12 MHZ
    set_scenario_clk(CLK_12MHZ); set_scenario_voltage(PS_1);
}
```

Figure 5: Changes to the code after simple loop transforms

the slack available with the sum of loopcnt and the corresponding sleep wakeup time until it finds a solution. If no solution is found the system stays idle until the arrival of the next frame. For simplicity, the AMU_2 mechanism is not shown in Figure 5. The AMU_2 is executed inline after the nested for-loops in Figure 5, and the selected sleep mode is then used in the RTH mechanism directly. The AMU_2 mechanism is thus a separate inline function added in addition to the AMU_1 function in our AMU part in Figure 5.

```c
static inline void AMU_2(volatile int *scenario){
  if(*scenario == DVFS_40MHZ_1_8V){ // Current scenario = PS0
    if( ((g_rts_data)[0] + (g_rth_lc)[5]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_RET1; // loopcnt + RET1 wakeup < Slack
    else if( ((g_rts_data)[0] + (g_rth_lc)[4]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_RET0; // loopcnt + RET0 wakeup < Slack
//...
    else if( ((g_rts_data)[0] + (g_rth_lc)[0]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_SLEEP00; // loopcnt + SLEEP00 wakeup < Slack
    else
      (g_rts_data)[1] = (long) PS0_RUN0; // No RTH
  }else if(*scenario == DVFS_12MHZ_1_2V)\{ // Current scenario = PS1
    if( ((g_rts_data)[0] + (g_rth_lc)[10]) < TOTAL_FRAME_CYCLES_12MHZ )
      (g_rts_data)[1] = (long) PS1_RET1; // loopcnt + RET1 wakeup < Slack
//...
    else if( ((g_rts_data)[0] + (g_rth_lc)[6]) < TOTAL_FRAME_CYCLES_12MHZ  )
      (g_rts_data)[1] = (long) PS1_SLEEP10; // loopcnt + SLEEP10 wakeup < Slack
    else
      (g_rts_data)[1] = (long) PS1_RUN1; // No RTH
  }
}
```

Listing 2: AMU_2 structure

8. Experimental results

In our experiments we simulate a realistic video sequence of 500 seconds using the experimental setup presented in Section 5, assuming the video sequence can be processed with 25 ARM Cortex M4 cores in parallel in order to meet the timing requirements for a frame rate of 25 frames per second (fps). The parallel cores are assumed able to process each frame individually with negligible synchronization overhead. This solution requires buffering and software pipelining of the frames to maintain forward frame dependency requirements. Other researchers have suggested similar approaches where their focus targets the parallelization of the H264/AVC encoder [8], [9]. We do not target general high-performance multi-core platforms. Instead, we aim at a subset of applications with non-interacting concurrently operating tasks. Moreover, we target the embedded multi-core domain, where cache coherence and complex bus protocols are avoided by limiting the functionality of the cores. In particular, we limit ourselves to totally independent cores working on tasks that do not have any interaction, where we believe this extrapolation is valid.

Our scenario implementation is independent of this parallelization, and can be used on a faster processor for a single-core solution. After each frame is processed, we force the core into a low power retention mode, where the measured energy consumption is negligible. Before entering retention mode, a configurable timer is set to wake up the system just before the next frame arrives.

Our video sequence is measured using all three network measurements in Figure 3. To process a frame, we specify a threshold requirement, bw_th, for the available bandwidth to be equal to at least five times (5x) the required bandwidth for a frame rate of 25 fps with a particular frame size. This threshold allows other activity to continue on the network, and avoids using up all the available bandwidth. With lower thresholds, the limited number of available voltage levels in the SAM4L board would always force the system to use the most power consuming voltage setting. The 5x threshold thus enables us to demonstrate how our methodology exploits the dynamism in the available bandwidth. Our simulated 500-second video sequence is created by combining different frame sizes from Table 3 in a sequence that fits with the available bandwidth of the measured networks in Figure 3, and our threshold requirements.

8.1. Energy reductions compared to a RTH approach without system scenarios

The application code is compiled in a conventional way with compiler optimization parameter O3. When we optimize the application code with the O3 configuration, we assume it is a good enough optimization for the exploitation step in the system scenario design methodology presented by Gheorghita et al. [3]. The energy consumption is measured in real time using our experimental setup, with and without the scenario mechanism. The overhead energy consumption of our scenario mechanism is measured by forcing the scenario mechanism to select PS0 each time a scenario switch takes place. This approach would not reduce the energy, but would trigger the scenario mechanisms in our AMU and PAM modules. The results are then compared to the energy consumption without using the scenario mechanism. The difference equals the scenario mechanism overhead.

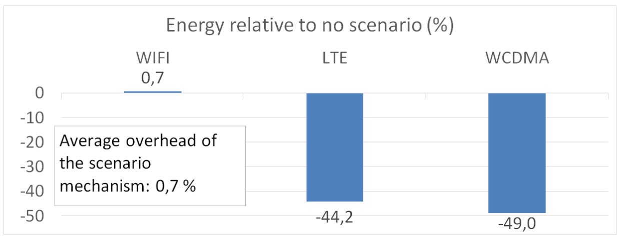


Figure 6: Energy consumption relative to no system scenarios.

Figure 6 shows the energy reductions relative to the energy consumed by a no scenario implementation utilizing a race to halt (RTH) setup in each of the measured networks of Figure 3. A retention mode is used when the system is halted, which consumes 4.4 μW. The average energy overhead of our scenario mechanisms are measured to be 0.7% after our improvements discussed in Section 7.2. We consider this overhead negligible.

The lowest energy consuming situation occurs when all frame sizes are suitable for the PS1 configurations, such as for the WCDMA network. We obtain in this situation up to 49% energy reduction as compared to a RTH solution without the use of the scenario mechanism. These results include the average scenario mechanism overhead of 0.7% energy increase for the 500-second sequence.

When the available bandwidth is higher than our threshold for our largest frame size, then the scenario mechanism is not needed. It simulates our most energy-consuming situation, where the system is running constantly in the PS0 platform configuration. In this situation the energy consumption does not increase more than the measured scenario overhead.

The LTE results are interesting because the energy reduction for this network is determined by how long the system is in the PS1 configuration. In our measurement, the platform runs with the PS1 configuration 93.8% of the time for this LTE network measurement (with bw_th = 5x), and runs in the PS0 for the remaining sequence. The energy savings would decrease if the platform runs in PS0 for longer time periods.

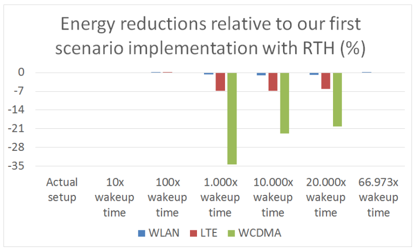
Our results meets Hillestad's [6] requirement for perceptual video quality, as mentioned in Section 1, while maintaining the frame rate.

8.2. Exploiting available sleep modes based on remaining available slack time

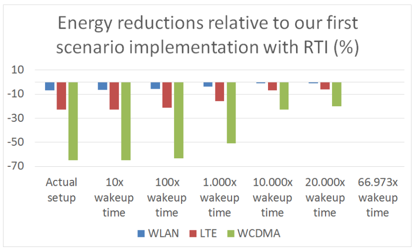
The energy overhead of our scenario mechanism is measured using our streams with and without the AMU_2 and its corresponding lookup tables for the system scenario sleep mode management. The lookup tables for the scenario sleep mode management are based on Tables 1 and 2. The added overhead of the AMU_2 is measured to be less than 0.1% and therefore we consider the added overhead negligible. With our experimental setup all frames, even the largest one, are processed and finished with enough slack time for our AMU_2 to select the lowest power consuming sleep mode. The sleep mode scenarios consequently does not give any energy reduction. If, on the other hand, the slack time of different frame sizes would be in the same range as wakeup times, we would benefit from selecting the optimal sleep mode. To illustrate this claim, we have simulated our recorded streams assuming the wakeup time of each sleep mode were a factor N slower, where N equals 1x, 10x, 100x, 1.000x, 10.000x, 20.000x and 66.973x. If we were actively using other types of memories (e.g. off-chip memories) then the wakeup times for these memories are more likely to be at least 1.000 times slower, which makes our assumptions realistic. If the slack time would be significantly shorter than what we have in our experiment, such that our most power efficient sleep mode could not be selected, then we also expect similar energy reductions. If the wakeup time is too long, which is the case if N > 66.973x, the system would never be able to go into a sleep mode. Based on this we see that our application benefits from the AMU_2 if its task slack times dynamically changes between different processing intervals and between different sleep mode wakeup times.

Without AMU_2 the RTH mechanism cannot adapt the sleep mode selection according to dynamically changing slack times. Instead the least power consuming sleep mode that still is able to wake up in time after the largest frame size has to be selected in all situations. This sleep mode will be increasingly more power consuming when N increases, and for N > 66.973x the wakeup times of all sleep modes will be too long. The AMU_2 mechanism exploit lower power consuming sleep modes when smaller frames are processed, instead of only selecting the safest sleep mode. The estimated energy reductions compared to not having the AMU_2 (and corresponding lookup tables) available are shown in Figure 7a. Figure 7b shows the energy reductions compared to a Race-To-Idle (RTI) setup where the processor does not go to sleep or halt after each processed frame.

Figure 7 clearly shows that we would start to get significant energy reductions if the wakeup times for our sleep modes were approximately 1.000 times slower. In this case, our AMU_2 approach would result in up to 34.3% energy reduction. The effect decreases when N increases since fewer sleep modes are within the range of the slack times. In our simulation, we observe that if the wakeup times were more than 66.973 times slower, then even the sleep mode with the shortest wakeup time would become too slow. In this situation our processor would stay idle until the next set of frames arrive. When the wakeup times are only 100 times slower, we observe a slight increase in energy consumption. This effect is caused by the fact that our AMU_2 mechanism selects sleep mode sleep03 over sleep02 because it has lower power consumption. On the other hand, the sleep03 wakeup time is significantly larger than the sleep02 wakeup time as shown in Table 1, and this large difference affects the total consumed energy consumption. If the wakeup times would be more evenly scaled between the different sleep modes, then this effect would not appear.



(a) Sleep scenarios compared to scenario Race-to-Halt (RTH).



(b) Sleep scenarios compared to Race-to-Idle (RTI).

Figure 7: Potential improvements in energy consumption with slower wakeup times.

8.3. Introducing more DVFS states into our scenario mechanisms

Our SAM4L board configuration capability is limited to two voltage states. If more voltage states were available, which is the case for modern Intel processors for example, the energy consumption is expected to decrease further since each frame size could correspond to a separate DVFS setting.

According to Ramalingam et al. [29], the Sakurai-Newton (SN) delay approximation is a widely used closed-form delay metric for the CMOS gates because of simplicity and reasonable accuracy. This approximation is good when the supply voltage is significantly larger than the threshold voltage. We use this delay approximation model to get the relationship between maximum voltage and frequency for a single gate as shown in Equation 4, where V_{DD} is the supply voltage, V_t is the threshold voltage, C is the load capacitance, and $k = (\frac{W}{L})\mu_n C_{ox}$ (logic design and fabrication characteristics factors). The α-power dependence is the velocity saturation index, which is a parameter between 1 and 2. According to Gonzalez et al. [30], the velocity saturation index is likely to be 1.3-1.5 in a 0.25μm technology.

$$\frac{1}{\tau} \approx f \approx k \cdot \frac{(V_{DD} - V_t)^\alpha}{C \cdot V_{DD}} \tag{4}$$

To determine the necessary change in operating frequency when voltage is scaled, we observe that Equation 4 is close to linear when $\alpha \approx 1.5$. Measured voltage-frequency curves from a 32nm bulk process [31] and a 28nm FD-SOI process [32] also show a roughly linear relationship between voltage and frequency (the frequency scale in [31] is logarithmic). Therefore, we will assume a linear frequency-voltage dependence when the supply voltage is significantly higher than the threshold voltage.

To demonstrate the added gain of more DVFS settings, we have extrapolated our available DVFS settings with six evenly spaced voltage and frequency levels based on our assumption of linear frequency-voltage dependence, ranging from 1.05 V to 1.8 V and from 6 MHz to 40 MHz, respectively.

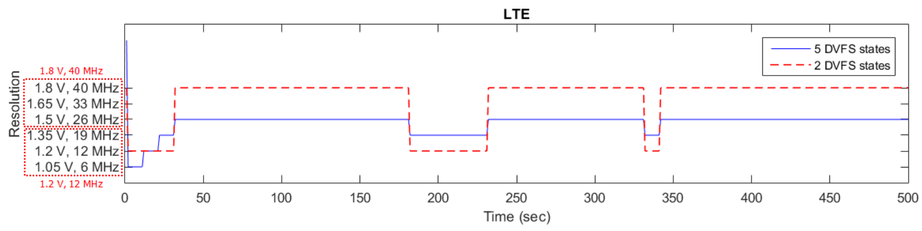


Figure 8: DVFS state transitions during the 500-second video clip.

The corresponding scenario assignment for the different frame sizes are shown in the scenario* column in Table 3. Figure 8 shows all scenario configurations from scenario 0 (1.8V/40MHz) to scenario 5 (1.05V/6MHz). We simulated our video sequence by manually applying the new scenario selection based on the LTE bandwidth measurement shown in Figure 3, and calculating the dynamic energy. A lower threshold value, equal to four times the required bandwidth (bw_th = 4x instead of 5x), is selected to utilize better our extrapolated DVFS settings. The increased number of DVFS settings allows the system to use a lower threshold for the bandwidth requirement to process a particular frame. Our simulated result based on extrapolated DVFS settings is shown in Figure 8 for the LTE network. From this figure, we observe that the system switches to a higher frequency five times, and it scales down three times. Our existing solution with two DVFS settings would switch back and forth between two settings 6 times in total. Taking Equation 1 and the energy overhead we measured from Table 2 into consideration, our extrapolated results indicate that the dynamic energy is now reduced 59% compared to not having scenarios. A multi-core extension of our scenario mechanisms needs to address cache coherence and on-chip communication challenges in addition to the scenario selection at run-time. This extension is out of the scope of this paper.

9. Conclusion

Our framework for systems scenario based designs (FSS) is combined with race to halt (RTH) and dynamic voltage and frequency scaling (DVFS) in order to achieve significant energy reductions for an H264/AVC encoder control structure. We measure our results on a modified SAM4L board from Atmel, which enables online DVFS tuning with switching times comparable to state-of-the-art switching times.

System scenarios are identified based on data-dependent control variables, varying with the input frame size. These scenarios are predicted by using inline estimation functions for the expected execution time. These functions are executed at run time, and are compared to a predefined set of ranges for the scenario prediction with minimum overhead. Our switching mechanism then re-configures the platforms voltage and frequency accordingly.

We measure the available bandwidth in WIFI, LTE, and WCDMA networks for 500 seconds. A video stream consisting of different frame sizes is then composed based on the measured network data and a threshold requirement. Compared to a no system scenario solution with RTH we achieve up to 49% energy reduction for the encoded streams, including a 0.7% average energy overhead, which is negligible. This average energy overhead is a result of optimal tuning and simple loop transformations, compared to our initial approach using conservative tuning having 8.5% average energy overhead.

The perceptual video quality and frame rate is maintained, and we show how we could reduce the energy consumption even further if a few more voltage levels are available as compared to only two available voltage levels.

We also show how existing sleep modes can be exploited in order to gain up to 34.3% energy reduction if the sleep wakeup times were 1000 times slower than our actual board. Our assumption for 1000 times slower wakeup times are representative for solutions dependent on off-chip memories or other dependent modules that require long wakeup times. We expect this assumption also to hold if the slack time of the processed task is in the range of the sleep wakeup times.

Furthermore, we expect similar results to be obtained by other applications with similar characteristics. The techniques for scenario identification and switching presented here enables efficient use of our framework for system scenarios, especially for applications where the dynamic behavior is determined by many-valued data variables.

References

[1] D. Wu, et al., Scenario-based system design with colored petri nets: an application to train control systems, Software & Systems Modeling (2016) 1–23.

[2] Z. Ma, Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms, Springer, 2007.

[3] S. V. Gheorghita, et al., System-scenario-based design of dynamic embedded systems., ACM Trans. Design Autom. Electr. Syst. 14 (1).

[4] Y. Yassin, et al., System scenario framework evaluation on efm32 using the h264/avc encoder control structure, in: 2015 European Conference on Circuit Theory and Design (ECCTD), 2015, pp. 1–4.

[5] M. A. Awan, et al., Race-to-halt energy saving strategies, Jour. of Systems Architecture 60 (10) (2014) 796 – 815.

[6] O. Hillestad, Evaluating and enhancing the performance of ip-based streaming media services and applications, Ph.D. thesis, Norwegian university of science and technology, doctoral thesis (2007).

[7] G. Rao, et al., Real-time software implementation of h.264 baseline profile video encoder for mobile and handheld devices, in: 2006 IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP 2006 Proc., Vol. 5, 2006, pp. V–V.

[8] Z. Zhao, et al., A highly efficient parallel algorithm for h.264 video encoder, in: Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on, Vol. 5, 2006, pp. V–V.

[9] S. Sun, et al., A highly efficient parallel algorithm for h.264 encoder based on macro-block region partition, in: R. Perrott, B. Chapman, J. Subhlok, R. de Mello, L. Yang (Eds.), High Performance Computing and Communications, Vol. 4782 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 577–585.

[10] C. Lee, et al., Mediabench: a tool for evaluating and synthesizing multimedia and communications systems, in: Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on, 1997, pp. 330–335.

[11] Y. Yassin, et al., Dynamic hardware management of the h264/avc encoder control structure using a framework for system scenarios, in: 19th EUROMICRO Conference on Digital System Design (DSD'16). IEEE, 2016, 2016, pp. 1–8.

[12] Atmel, Sam4l xplained pro user guide (2014).

[13] Y.-H. Lee, et al., Scheduling techniques for reducing leakage power in hard real-time systems, in: Real-Time Systems, 2003. Proc. on 15th Euromicro Conference, 2003, pp. 105–112.

[14] S. Irani, et al., Algorithms for power savings, ACM Trans. Algorithms 3 (4).

[15] L. Niu, et al., Reducing both dynamic and leakage energy consumption for hard real-time systems, in: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '04, ACM, New York, NY, USA, 2004, pp. 140–148.

[16] J. K. Rai, et al., Machine learning based performance prediction for multi-core simulation, in: Proceedings of the 5th International Conference on Multi-Disciplinary Trends in Artificial Intelligence, MIWAI'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 236–247.

[17] S. V. Gheorghita, et al., Automatic scenario detection for improved wcet estimation, in: Proceedings of the 42Nd Annual Design Automation Conference, DAC '05, ACM, USA, 2005, pp. 101–104.

[18] S. V. Gheorghita, et al., Scenario selection and prediction for dvs-aware scheduling of multimedia applications., Signal Processing Systems 50 (2) (2008) 137–161.

[19] E. Hammari, et al., Identifying data-dependent system scenarios in a dynamic embedded system, The International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'12, Las Vegas, USA.

[20] M. Palkovic, et al., Dealing with variable trip count loops in system level exploration., in: Proc. of the 4th Workshop on Optimizations for DSP and Embedded Systems, 2006, pp. 19–28.

[21] M. Damavandpeyma, et al., Throughput-constrained dvfs for scenario-aware dataflow graphs, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, 2013, pp. 175–184.

[22] S. Mohapatra, et al., Integrated power management for video streaming to mobile handheld devices, in: Proceedings of the Eleventh ACM International Conference on Multimedia, MULTIMEDIA '03, ACM, New York, NY, USA, 2003, pp. 582–591.

[23] P. Van Dyke, et al., Apparatus and method for resizing an image, US Patent 7,733,405 (2010).

[24] S. Sesia, I. Toufik, M. Baker, LTE-the UMTS long term evolution, Wiley Online Library, 2015.

[25] E. Dahlman, et al., Wcdma-the radio interface for future mobile multimedia communications, Vehicular Technology, IEEE Transactions on 47 (4) (1998) 1105–1118.

[26] Iperf, networking with iperf. [Online].

[27] H. K. Flora, et al., An investigation on the characteristics of mobile applications: A survey study, International Journal of Information Technology and Computer Science (IJITCS) 6 (11) (2014) 21.

[28] National Instruments, Ni mydaq measurement board (2015).

[29] A. Ramalingam, et al., Robust analytical gate delay modeling for low voltage circuits, in: Asia and South Pacific Conference on Design Automation, 2006., 2006, pp. 6 pp.–.

[30] R. Gonzalez, et al., Supply and threshold voltage scaling for low power cmos, IEEE Journal of Solid-State Circuits 32 (8) (1997) 1210–1216.

[31] S. Jain, et al., A 280mv-to-1.2v wide-operating-range ia-32 processor in 32nm cmos, in: 2012 IEEE International Solid-State Circuits Conference, 2012, pp. 66–68.

[32] B. Zimmer, et al., A risc-v vector processor with tightly-integrated switched-capacitor dc-dc converters in 28nm fdsoi, in: 2015 Symposium on VLSI Circuits (VLSI Circuits), 2015, pp. C316–C317.