**LD-DVS: Load-Aware Dual-Speed Dynamic Voltage Scaling**

Christian Poellabauer, Dinesh Rajan, and Russell Zuck  
Computer Science and Engineering,  
University of Notre Dame  
Notre Dame, IN 46556  
E-mail: {cpoellab,dpandiar,rzuck}@cse.nd.edu

**Abstract**

The goal of Dynamic Voltage Scaling (DVS) is to maximize the energy savings while ensuring that applications’ real-time requirements are met. Accurate predictions of task run-times are necessary to compute an appropriate CPU frequency that achieves high energy savings, avoids deadlines misses, and reduces the overheads caused by frequent changes between different frequency levels. This paper experimentally explores an architecture based on the XScale PXA255 processor and shows that workload-awareness is not only required for accurate predictions of utilization, but also that in systems with a discrete number of frequency levels, the energy savings achieved by existing dual-speed DVS approaches (where an optimal theoretical CPU speed is computed and then approximated by choosing the two neighboring discrete speed levels) are suboptimal. As a consequence, this work introduces an online approach to dual-speed DVS that formulates a model for speed selection based on the workload characteristics of the current task set and computes a frequency pair that yields the best possible energy savings for a given task set and workload.

1 INTRODUCTION

**Background.** In recent years, mobile devices have become highly integrated into the daily lives of their users. However, a fundamental deficiency of these devices is their short battery lives, which is becoming an ever larger hindrance to their continued pervasiveness. This demands the use of efficient energy and power management techniques such as power-aware scheduling policies (Kim et al., 2002b; Mejia-Alvarez et al., 2002; Poellabauer and Schwan, 2002; Saewong and Rajkumar, 2003), low-power modes for disks and networks (Chandra and Vahdat, 2002; Helmbold et al., 1996), or energy management techniques for communication in mobile systems (Monks et al., 2001; Poellabauer and Schwan, 2004a,b; Qiao et al., 2003).

One approach to extend the battery life of mobile devices is to employ a speed regulation technique known as Dynamic Voltage Scaling (DVS) (Flautner et al., 2001; Hsu and Kremer, 2000; Krishna and Lee, 2003; Lorch and Smith, 2001; Pouwelse et al., 2001; Saewong and Rajkumar, 2003; Simunic et al., 2001; Zhu and Mueller, 2004). DVS techniques take advantage of the fact that the processor power consumption is proportional to the product of frequency and the square of voltage, and thus substantial power savings can be realized by modulating the voltage and frequency combination of the processor (Burd and Brodersen, 1995). Essentially, DVS algorithms attempt to maintain CPU utilization close to 100%, while taking into account the timing constraints of the processes executed on the device at any given time. The use of DVS has to be carefully chosen such that applications’ real-time and Quality-of-Service (QoS) requirements are satisfied, while limiting the energy consumption and increasing the battery life time. Most modern mobile processors (such as Intel’s XScale and Transmeta’s Crusoe chips) offer built-in support for voltage scaling to address the energy constraints of their devices.

This paper investigates the use of DVS on an embedded platform based on the XScale PXA255 processor, which supports a limited number of discrete frequency levels. Recent DVS techniques base the frequency/voltage computations used by the scheduler on their predictions of future task run-times (Pillai and Shin, 2001; Zhu and Mueller, 2004). However, these approaches ignore that applications’ run-times also depend on I/O, memory accesses, or interrupts. Inaccuracies in run-time predictions can therefore lead to missed deadlines, suboptimal energy savings, or large overheads due to frequent re-computations and adjustments of the frequency and voltage.

The work presented in this paper is based on **workload-awareness**, i.e., inaccuracies in predicting task run-times...
are addressed by considering their sources in the prediction process. One of these sources is utilization of memory resources. Memory-bound applications, including image/video processing and scientific applications, require frequent accesses to memory, which contribute substantially to their total run-times, particularly in systems where DVS not only changes the CPU’s clock frequency, but also the frequency of the bus linking CPU and memory. Therefore, we capture the memory overheads experienced by memory-bound tasks and then use these overheads to better predict task run-times. This is achieved by monitoring a task’s cache miss rate, which determines the memory access rate experienced by this task. These cache miss rates are used as feedback to the DVS algorithm, which then computes the frequencies and voltages to be used for task execution.

Load-Aware Dual-Speed DVS. Further, practical systems offer only a limited number of frequency/voltage settings; as a consequence, when computing a theoretical optimal speed setting, typically the next higher speed setting has to be selected to avoid CPU utilizations greater than 100%. Dual-speed approaches to DVS (Seth et al., 2003; Xie et al., 2003; Zhang and Chanson, 2002) address this limitation by calculating an optimal theoretical frequency setting and choosing between the two discrete speed settings (e.g., the two neighboring frequency levels) to approximate the theoretical result. Similarly, other DVS algorithms such as presented by Pillai and Shin (2001), Saewong and Rajkumar (2003), and Zhu and Mueller (2004) compute the currently optimal speed setting more frequently (e.g., on a per-task level), resulting in frequent speed changes, but with the same goal of approximating the theoretical optimum.

Through our experiments, we compare the energy savings of different frequency combinations under different workloads, showing that the selection of the speed pair depends on the current workload. A strategy considering only the two neighboring frequency levels does not always translate to the best possible approach that would yield the highest energy savings. While the I/O-dependency of DVS has been addressed in previous work, this paper introduces an online approach to dual-speed DVS that bases the speed selection on the workload characteristics of the current task set and frequently recalculates the frequency pair selection to ensure maximum energy savings.

Summary of Contributions. The main contributions of this work are the experimental study of the effects workload changes have on voltage scaling, the insight that the speed selection in dual-speed DVS algorithms has to consider workload characteristics, and the design, implementation, and evaluation of an online workload-aware dual-speed algorithm.

2 RELATED WORK

Previous efforts on DVS for real-time systems (e.g., as described by Pillai and Shin (2001) and Saewong and Rajkumar (2003)) rely on worst-case task execution times (WCET), whereas we consider average execution times and monitor dynamic cache miss rates to accurately predict future execution times. There has been substantial prior work that examines the effects of DVS on cache performance. Marculescu (2000) evaluated the energy savings due to the reduced cache miss penalties at lower clock frequencies. Simulation results with a SimpleScalar simulator show that a substantial reduction in energy consumption can be achieved with minimal performance degradation. Compiler-time techniques have exploited these stall periods, e.g., Hsu and Kremer (2000) implement a compiler-directed dynamic voltage scaling algorithm that achieves energy savings of up to 55%, with only a 6% performance penalty.

Varma et al. (2003) investigated the use of feedback loops to refine approaches such as PAST (Weiser et al., 1994) by considering the rate of change in a system’s processing requirements. Weissel and Bellosa (2002) also distinguish between processing and memory access, but their work is limited to frequency scaling, where each task operates at its own speed. With voltage scaling, approaches that compute frequency and voltage pairs over the entire task set are preferable due to the reduced changes in speed settings and thereby reduced overheads. The FAST approach (Seth et al., 2003) focuses on static timing analysis for simple in-order single issue pipeline. Although the authors argue that static timing analysis is preferable over dynamic approaches (because they do not provide any worst-case execution time guarantees), the results for our feedback-based approach on an out-of-order completion processor show that ‘real’ applications (e.g., gzip) have cache miss rates that do change, but by no means in a random way.

Further, Choi et al. (2004) utilize cache miss feedback based on performance counters as shown in our work, but focusing on best-effort tasks. For example, to limit the overheads of frequent voltage changes, they use a time quantum of 60ms, which prohibits rapid changes often needed in real-time systems. Our approach therefore can change voltages at any time, but it uses a Schmitt-trigger-style function to limit the number of voltage changes and the overheads associated with these changes.

In previous work, DVS algorithms calculate a single desired voltage-frequency pair based on the timing constraints and utilization of the system (Choi et al., 2004; Kim et al., 2002a; Pillai and Shin, 2001; Pouwelse et al., 2001; Saewong and Rajkumar, 2003; Yuan and Nahrstedt, 2004). With processors that support only a limited number of operating frequency levels, there can be a significant difference between the desired and the actual operating frequency and
this error translates directly to a loss in any potential energy savings. As a consequence, typical DVS algorithms frequently need to recompute the desired voltage-frequency combinations to achieve optimum energy utilization. Sae-wong and Rajkumar (2003) extensively studied the effects of a finite number of operating frequencies on the energy conservation achieved by different DVS techniques. Their study proposed an increase in the number of supported operating frequencies in order to achieve minimum energy quantization loss that results from processors having to run at the next higher frequency when a desired frequency is not available. However, other efforts (Bini et al., 2005; Seth et al., 2003; Xie et al., 2003; Zhang and Chanson, 2002) show that the energy quantization loss can still be significantly reduced by intelligently employing a dual-speed approach to voltage scaling.

Further, most DVS solutions ignore the effect the current workload may have on the potential energy savings offered by the different voltage-frequency levels allowed in a processor. Fan et al. (2003) present a relationship between the power consumption of memory and the energy savings achieved by DVS techniques. This work also provides a detailed analysis of the effect of cache miss rates on the performance of DVS algorithms. But it does not provide any implementation of a policy to address the concerns raised due to ignoring memory performance and power consumption. Jejurikar and Gupta (2004) consider the external factors of leakage power consumption and the standby energy consumption of peripheral devices in computing an optimal voltage-frequency combination. Their work proposes an efficient mechanism for slowing down tasks based on both leakage power and standby energy.

Finally, efforts similar to our work were made by Choi et al. (2004), Poellabauer et al. (2005), and Zhu and Mueller (2004), where DVS approaches were implemented that consider the effects of external memory accesses in computing a single optimal frequency setting. Our work extends on this effort by employing a dual-speed approach to achieve the optimum frequency level, based on the insight that different selections of two speeds (from a discrete set of available speeds) result in different energy consumptions.

We thereby propose an approach that exploits existing DVS techniques by dynamically computing two voltage-frequency combinations and a ratio of the run-times between these two states (determining how long the processor has to operate in each state). However, the key novelty of our work is that we employ dynamic feedback from the current utilization and resource usage of tasks such as cache miss rates, to dynamically adjust the run-time ratio and, if required, the frequency-voltage settings, as opposed to previous static or compile-time solutions (Seth et al., 2003; Xie et al., 2003).

3 DYNAMIC VOLTAGE SCALING

3.1 Terminology and Scheduler Model

In order to simplify our discussion in future sections, we briefly discuss the following two terms: P-state, and E-state. The P-state of a system, or Power-state, is identified by the \((f, V, x)\)-level under which the system is currently operating. It is referred to as a P-state because of the direct relationship that exists between power, frequency, and voltage of operation of the CPU. The energy consumed by the system is directly related to the time for which the system operates at a particular P-state. We use the term E-state of a system to describe an \((f, V, x)\)-level, where \((f, V)\) is the system’s P-state and \(x\) is the amount of time (in percent) the system operates in that particular P-state. The processor speed is proportional to the clock frequency, i.e., \(S \propto f_c\). The power consumed for CMOS chips is expressed as: \(P \propto f_c \cdot V^2\), where \(V\) is the voltage supplied to the processor (Burd and Brodersen, 1995). With DVS, both the frequency and the voltage can be reduced if idle CPU resources are available, resulting in a performance-energy tradeoff, i.e., the energy consumption is reduced, while the latencies are increased.

The focus of our work is on aggressive energy conservation, i.e., instead of worst-case execution times (WCET), the proposed approach predicts actual execution times, where the accuracy of such predictions determines the amount of achievable energy savings and the number of deadlines met. Therefore, the remainder of this paper considers soft real-time systems, i.e., occasional deadline misses do not cause the system to fail.

We consider task sets of periodic tasks, where each task \(i\) has a period \(T_i\) associated and the end of the current period is the deadline. Further, each task \(i\) is guaranteed \(C_i\) time units for each period \(T_i\), and the scheduler uses Earliest Deadline First (EDF) among all currently ‘eligible’ tasks (i.e., tasks that have not yet executed for their time slices in their respective periods). Non-periodic tasks such as compression tools, compilers, etc., are represented as periodic tasks by guaranteeing them a certain amount of progress \((C_i)\) during each period of \(T_i\). The actual execution time of the most recent task invocation is \(C_i' \leq C_i\) and the actual execution time can vary from period to period. The speed of a CPU is slowed down to take advantage of rest utilization (i.e., utilization bound \(U_{max}\) minus the current utilization \(U_{system}\)) and the clock frequency is recomputed whenever the system utilization changes, e.g., when tasks join or leave the run queue. The current utilization of all tasks is computed with: \(U_{system} = \sum C_i' \cdot F_i\) and serves as the basis for our voltage scaling approach. That is, the most recent actual execution times of all tasks are used to predict the future utilization of the system.
The remainder of this section discusses both workload-aware DVS and dual-speed DVS, followed by an integration of both in Section 4.

3.2 Load-Aware DVS

For each supported clock frequency \( f_n \), a scaling factor \( k_n \) can be obtained by executing some processing-intensive code at both the default frequency (maximum) \( f_{max} \) and \( f_n \) and dividing their measured run-times \( C_n \) and \( C_{max} \), i.e.,

\[
k_n = C_n / C_{max}.
\]

This is repeated for each available clock frequency (or core voltage) for a given processor and the results can be stored in a table accessed by a DVS algorithm. The goal of frequency scaling is to get as close to 100% utilization as possible, i.e.,

\[
U_{100\%} = \frac{\sum C'_{task}}{\sum f_{task}},
\]

where \( k' \) is the yet unknown scaling factor. To guarantee that best-effort tasks do not starve, we can replace \( U_{100\%} \) with \( U_x \) where \( x < 100\% \). The value of \( k' \) can then be determined with:

\[
k' = \frac{U_x}{\sum f_x}.
\]

The resulting \( k' \) is compared to the table containing the previously computed scaling factors and the scaling factor \( k_n \) closest to \( k' (k_n \leq k') \) is selected. The clock frequency is then adjusted to the frequency \( f_n \) (\( f_n \geq f' \)).

Note that there are approaches which compute different scaling factors for each individual application (intra-task approaches as opposed to inter-task approaches), but such approaches typically incur higher overheads caused by more frequent transitions between different speed levels (Hsu and Kremer, 2000; Lorch and Smith, 2001). Nevertheless, the solution introduced in this paper could also be applied to these approaches.

Memory-Awareness. Intuitively, the scaling factor \( k' \) would be the ratio of the clock frequencies. However, various factors, including cache misses penalties, context switching, and I/O requests cause a change in \( k' \).

Figure 1 shows the performance of three applications, relative to the maximum frequency (i.e., the inverse of the execution times relative to the execution time at the maximum frequency). On the x-axis, we display the different frequencies supported by an XScale-based device: the first number depicts the core clock frequency (ranging from 99MHz to 398MHz); the second number is the bus frequency (ranging from 50MHz to 196MHz). This results in 5 different frequency settings (the fourth and fifth settings differ only in the bus frequency). The three programs selected are dcache-miss, a C program written specifically to generate data cache misses by performing memory accesses on a large array; gzip, the popular file compression application; and cjpeg, a JPEG image compressor. The theoretical line shows the expected ratio, assuming that the execution time was linearly related to the CPU frequency. With memory-intensive programs like gzip and dcachemiss, execution times differ substantially from the theoretical line, since the execution times of these programs are more dependent on the bus frequency than the CPU frequency. The third application, cjpeg, is less memory-intensive and has an expected ratio closer to the theoretical line compared to the two other applications. Intuitively, each process by itself or each combination of different processes could require a different \( k' \) factor. However, the approach addressed in this paper uses one \( k' \) factor for all processes currently on the run queue. If a simple linear model were used to determine the \( k' \) factors, it would underestimate the utilization at lower clock frequencies, resulting in deadline misses, and overestimate the utilization at higher clock frequencies, resulting in wasted energy. Note that the lines cross at 298MHz (99MHz) which is due to the increasing CPU speed, while the bus speed remains constant. That is, at CPU speeds of 99 and 199MHz, memory-intensive applications experience larger penalties in execution time than CPU-intensive applications. But with increasing CPU speeds (and constant bus speeds), the penalty due to the slower bus speed becomes less significant.

In summary, the differences in the execution times shown in Figure 1 underline the importance of considering cache miss rates in the DVS approach to accurately predict future task run-times. Note that these measurements have been performed on an XScale processor, a modern mobile scalar processor with a 7-stage pipeline and out-of-order completion. While modern processors can hide certain memory latencies, the hundreds of cycles resulting from DRAM accesses can cause significant performance losses (Wang et al., 2003). Therefore, we use cache misses
as indicator for the performance penalties incurred by memory accesses. The remainder of this paper will use the Memory Access Rate (MAR) to quantify the rate of cache misses. As opposed to the more commonly used cache miss rate, MAR is defined in Equation 1 as the ratio of data cache misses to instructions executed.

\[
MAR = \frac{\text{data cache misses}}{\text{instructions executed}}
\] (1)

The reason we rely on MAR as a measure of miss rates is due to the monitoring capabilities of the investigated architecture (described in Section 3.4).

Figure 2 shows gzip’s MAR plotted over time. The graph shows a behavior where overall the miss rate is rather constant, but can change dramatically as shown in the figure (e.g., different ‘phases’ of application execution). This was observed for a variety of applications. In this particular example, the MAR value can vary by a factor of over a hundred. Due to these variations, a single set of \( k' \) values would not be sufficient for DVS over the entire execution of gzip.

### 3.3 Dual-Speed DVS

With the dual-speed approach to DVS, two CPU speeds are computed (i.e., two E-states), and the CPU is switched between these two discrete speeds such that the optimal theoretical CPU speed is closely approximated. This will be the basis for our load-aware dual-speed DVS approach introduced later in this paper.

To illustrate the performance benefits of a dual-speed approach, let us consider an example using Figure 3. The desired \((f, V)\)-level for optimal energy savings is likely to fall between two discrete \((f, V)\) settings supported by the architecture. We assume that the optimal frequency is 260MHz for the first 150ms of the experiment and changes to 350MHz after that (e.g., because of workload changes). The dotted line in the figure represents this optimal frequency level. With a single speed approach, in order to ensure that the timing requirements of the task set are met, the processor selects the next higher frequency level (e.g., 298MHz instead of 260MHz). Unless the optimal speed matches exactly an available discrete speed level, this method will result in reduced energy savings. A better technique is to run at two adjacent frequency levels (198MHz, 298MHz) so as to achieve a frequency closer to the optimal frequency of 260MHz.

While the adjacent frequency levels seem to be a logical choice for a dual-speed DVS approach, the remainder of the paper shows that different combinations of two discrete \((f, V)\)-levels can be used to achieve the same average speed, however with different energy savings. Furthermore, the optimal combination of P-states (i.e., \((f, V)\)-levels) can change during run-time based on changes in utilization and type of workload (i.e., memory or I/O-intensive versus processing-intensive tasks). In our example above, different energy savings could be achieved by selecting frequency combinations of \((99, 298)\)MHz, \((99, 398)\)MHz, \((198, 298)\)MHz, or \((198, 398)\)MHz.

### 3.4 Sitsang Evaluation Platform

The Intel Sitsang evaluation board is based on the PXA255 XScale processor, supporting frequencies in the range of 99MHz to 398MHz and core voltages of 0.7V to 1.5V. The operating system used on the platform is a Linux 2.4.19 kernel specially ported for this architecture. We chose this platform because it represents the current and future set of embedded devices that combine many functions, capable of carrying out multiple tasks simultaneously. The Linux operating system readily supports speed and voltage

---

**Figure 2. Data memory access rate of gzip.**

**Figure 3. A dual-speed approach to DVS.**
Figure 4. Schmitt-trigger-style function mapping MARs to matrices of scaling factors.

Table 1 depicts the frequency-voltage combinations used in this paper for the Intel Sitsang evaluation board.

Table 1. CPU and bus frequencies supported by the Intel Sitsang PXA255 platform.

<table>
<thead>
<tr>
<th>P-state</th>
<th>CPU Freq.</th>
<th>Bus Freq.</th>
<th>Min. CPU Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99 MHz</td>
<td>50 MHz</td>
<td>0.8V</td>
</tr>
<tr>
<td>2</td>
<td>199 MHz</td>
<td>99 MHz</td>
<td>1.0V</td>
</tr>
<tr>
<td>3</td>
<td>298 MHz</td>
<td>99 MHz</td>
<td>1.1V</td>
</tr>
<tr>
<td>4</td>
<td>398 MHz</td>
<td>196 MHz</td>
<td>1.3V</td>
</tr>
</tbody>
</table>

Operating System Modifications. A field was added to Linux’s task_struct to keep a weighted average of the MAR for each process. The scheduler was modified to support DVS using the following algorithm:

```
1: prev->dcache_misses=(prev->dcache_misses/2) + pmu->dcache_misses;
2: prev->instructions=(prev->instructions/2) + pmu->instructions;
3: schedule();
4: inverse_mar=next->instructions/next->dcache_misses;
5: n=schmitt_function(inverse_mar);
6: M_n=freq_matrix[n];
7: new_freq=min_freq(M_n, cur_freq, utilization);
8: if (cur_freq != new_freq)
   9:   set_cpu_freq(new_freq);
10: reset_pmu_counters();
```

Lines 1-2 update the weighted average of the MAR for the most recently executing process (‘prev’). Line 3 invokes the CPU scheduler, selecting the next process. Once the next task is selected, we can compute \( n \), the \(-\log_2\) of the next task’s MAR, using the Schmitt-trigger-style function in Figure 4 (lines 4-5). This function avoids frequent changes in voltage and frequency, thereby avoiding the large overheads associated with these changes; for the test programs used in this paper we measured at most 3 voltage/frequency changes for the entire application run-time. The closest of
the 12 pre-computed frequency matrices, $M_n$, is selected from the lookup table (line 6). Line 7 uses the matrix of scaling factors to determine the minimum frequency needed to meet the deadline of a process (i.e., a simple matrix lookup), and if this frequency is different than the current frequency, the processor frequency is changed (lines 8-9). Finally, the XScale’s PMU counters used for monitoring the MAR are reset before the next process begins execution (line 10).

4 LOAD-AWARE DUAL-SPEED DVS

4.1 Load-Dependency of Speed Selection

The scaling factor $k'$ can typically range from 1.0 to 100.0 and is interpreted as the degree to which the system is under-utilized (i.e., a larger $k'$ represents a more under-utilized system). Figures 5-7 depict the normalized energy consumptions of the XScale processor (normalized to the default speed of (398, 1.3)) for various scaling factors.

Figure 5 represents the energy consumption when there are no memory accesses being made, while Figures 6 and 7 represent the energy consumed under two different rates of memory accesses (infrequent and frequent memory accesses, respectively). Each of the regions depicted in the figures represent the range between two supported operating frequencies. That is, region 1 represents the range between the two lowest frequencies 99 & 199MHz. Regions 2 and 3 map the ranges above the second lowest frequency, that is between 199 & 298 MHz and 298 & 398MHz, respectively.

The energy consumption presented here is measured for all possible combinations that could be chosen to approximate the desired computed frequency level that lies within a particular region, by switching frequencies between the two levels expressed in a combination. The graphs are plotted against the proportion of run-times that are normalized to the run-time at the lower frequency. For example, consider Figure 5. Region 1, as discussed earlier, represents the energy consumption for all those frequency combinations that can be used to achieve a frequency level between 99MHz and 199MHz. The region 1 has three plots that represent the possible frequency combinations that can be used to achieve any desired frequency between 99 and 199MHz. A value of 10% on the x-axis corresponds to running the task at the higher speed of a particular combination for 10% of its total execution time and 90% of the remaining time at the lower speed in that combination.

It can be observed from these graphs (Figures 5-7) that for a given region, running between its two end-point frequencies does not always achieve the maximum possible energy savings. As an example, consider Figure 5, where for region 2, choosing the frequencies 99MHz and 398MHz offers larger energy savings compared to choosing the two neighboring frequencies of the region - 198MHz and 298MHz. Single-speed DVS approaches would select the next highest frequency level (298MHz) when their computed optimal frequency falls within this region. On the other hand, dual-speed approaches switch between 198MHz and 298 MHz to approximate the optimal frequency. As can be inferred from the graphs, both ap-
memory access rates (MAR), to derive the current work-
load of the system. We employ feedback from the special
purpose registers providing information about the current
memory access rates. For example, for memory-
intensive tasks, the optimal frequency combination for re-
gion 2 is 198MHz and 398MHz (Figures 6 and 7) as op-
posed to 198MHz and 298MHz for processing-intensive
tasks, as shown in Figure 5. The reason for these changes
are that memory accesses of memory-bound applications
introduce significant latencies into the application run-
times, where these additional penalties differ for each of
the possible frequency combinations (e.g., because differ-
ent frequency settings use different bus speeds as shown in
Table 1). Thus, from the above data, we reach the conclu-
sion that an optimal DVS algorithm needs to consider the
load of a system in selecting a frequency-voltage setting for
largest energy savings. In the next section we introduce an
online feedback-based DVS approach to achieve this goal.

4.2 The LD-DVS Algorithm

In order to achieve optimal energy savings, a dynamic
voltage scaling approach should consider the current work-
load of the system. We employ feedback from the special
purpose registers providing information about the current
memory access rates (MAR), to derive the current work-
load of the system. This approach computes a new P-state
pair and a ratio of run times (i.e., how long to use the first
frequency and how long to use the second frequency) when-

The LD-DVS algorithm (Algorithm 1) computes a pair
of E-states and alternates the CPU speed between these
states. As a consequence, the optimal frequency is ap-
proximated, yielding the lowest possible energy consump-
tion for a system with discrete speed levels. The algorithm
initially determines the single optimal frequency level that
would lead to maximum processor utilization. All possi-le frequency pairs listed for the particular region are then
determined. Also, the ratios of run-times to be spent at
each of the two frequency levels in the possible frequency
combinations are determined. The algorithm then proceeds
to compute the normalized energy consumptions if run at
each of these possible frequency combinations. Note that
in the computation of the utilization values $U_{lowfreq}$ and
$U_{highfreq}$, we compute only the factor by which the system
would be under- or over-utilized if switched to that particu-
lar frequency. Also the achievable energy savings computed
by the algorithm for each of the possible frequency pairs are

![Figure 7. E-states for tasks with frequent memory accesses.](image-url)
Algorithm 1: Pseudocode for LD-DVS Algorithm

1: if change in utilization or workload (cache miss rate) then
2:   determine the single frequency that would achieve maximum utilization.
3:   determine the possible frequency combinations that can be used to achieve the optimum frequency level under the given workload.
4:   for each frequency pair do
5:     for the low and high frequency in the considered pair compute scaling factors \( k_1 = \frac{C_{\text{lowfreq}}}{C_{\text{currentfreq}}} \) and \( k_2 = \frac{C_{\text{highfreq}}}{C_{\text{currentfreq}}} \) respectively.
6:     calculate \( U_{\text{lowfreq}} = \frac{U_{\text{maxpossible}}}{U_{\text{current}}} \) and \( U_{\text{highfreq}} = U_{\text{maxpossible}} \) respectively.
7:     calculate \( x = \frac{U_{\text{lowfreq}}}{U_{\text{highfreq}} - U_{\text{lowfreq}}} \times 100 \) \{x indicates the proportion of time to be executed at the highest frequency of the E-state pair.\}
8:   for the low and high frequency in the considered pair compute scaling factors \( k_{\text{low}} = \frac{C_{\text{lowfreq}}}{C_{\text{maxfreq}}} \) and \( k_{\text{high}} = \frac{C_{\text{highfreq}}}{C_{\text{maxfreq}}} \) respectively.
9:   compute \( E_{\text{low}} \) and \( E_{\text{high}} \) as the energy that would be consumed if run individually at the low and high frequencies respectively.
10: calculate normalized energy consumption \( = (x \times E_{\text{high}} \times k_{\text{high}}) + ((100 - x) \times E_{\text{low}} \times k_{\text{low}}) \)
11: end for
12: for all frequency pairs do
13:   find the minimum of the computed normalized energy consumption values
14: end for
15: new frequency pair = frequency pair with minimum energy consumption
16: set processor frequency to run at the higher frequency in the computed pair
17: else
18:   if running at highfreq then
19:     update \( x_{\text{actual}} \)
20:     if \( x_{\text{actual}} > x \) then
21:       switch CPU to lowfreq
22:     end if
23:   else
24:     update \( x_{\text{actual}} \)
25:     if \( x_{\text{actual}} > (100 - x) \) then
26:       switch CPU to highfreq
27:     end if
28:   end if
29: end if

The frequency combination that yields the minimum normalized energy consumption is then chosen for further execution. The execution of the task set is switched between the two frequency levels in accordance with the ratio of run-times determined by the algorithm for the computed P-state pair. Thus, the output of the algorithm would be two E-states \((E_1, E_2)\), with \( E_1 = (f_1, v_1, x_1) \) and \( E_2 = (f_2, v_2, x_2) \), where \( x_1 \) and \( x_2 \) indicate the ratio of the run-times between \( E_1 \) and \( E_2 \) respectively such that \( x + y = 100\% \). If there is no change in the system utilization or workload, the algorithm would determine whether a switch between the frequency levels in the current frequency combination is required by comparing the current proportion of time spent at a frequency level with that of the corresponding computed proportion.

5 EVALUATION AND DISCUSSION

Overview. There are two primary types of implementation overheads that we consider. First, we examine the costs associated with switching between two P-states. Then we evaluate the overhead of executing the LD-DVS algorithm within the real-time scheduler.

In an effort to minimize the effects of tasks waiting/blocking for I/O operations in our calculations of the switching overhead, we employ an application tool \textit{dcache-miss} that allowed us to precisely model different data cache-
miss rates of tasks that would be executed in a typical mobile system. Utilizing dcache-miss also provides us with the additional benefit of minimizing the occurrence of job blocking conditions while applications wait for I/O. It also allows parameters such as task utilization and cache miss rates to be passed along with the individual tasks that are executed through this application.

In the following sections, we describe two types of experiments in which a pair of jobs were submitted to the real-time scheduler using dcache-miss. In the first type of experiments, both jobs are forced to yield similar cache miss rates thus producing an identical number of cache misses. However, each job was assigned a different utilization parameter. In this scenario, we would expect to see a re-calculation of P-state whenever there is a change in system utilization (whenever the second job is scheduled) or a change in cache miss rates.

Our second experiment employs the same two jobs and utilization requirements, but different parameters for cache miss rates were passed to dcache-miss to ensure frequent cache miss rate changes. Here, we expect to see more frequent P-state changes due to the greater variability in cache miss rates.

**Switching Overhead.** The time granularity of x86 Linux systems is known to be 10ms. Our LD-DVS algorithm is executed each time the scheduler runs in the Sitsang evaluation platform, thereby limiting the amount of overhead is introduced, even when task sets and workloads change frequently. Therefore, if we assume no blocking on I/O operations, the worst-case scenario would then be a change in the P-state every 10ms. Each P-state change results in about 500µs switching overhead, leading to a total overhead of 5%. But in all our experiments that simulated a host of real-time environments, we found P-state changes to be less frequent, typically resulting in overheads of 1 – 2%. The 500µs switching delay is primarily due to the fact that the available frequencies of the Sitsang evaluation platform are coupled to discrete voltage levels for the CPU. It is important to note that the switching overheads on this particular platform may be significantly higher than those experienced on more modern platforms and those that support several frequencies at the same core voltage.

For both experimental setups described above, we calculate the number of P-state changes that occurred within the first second of execution after both jobs were submitted to the scheduler. For the case where cache miss rates were similar, our algorithm reported about 20 P-state changes resulting in a switching overhead of 1%. In the case where cache miss rates were highly varied, a switching overhead of 1.25% was reported, resulting from 25 P-state transitions.

**Micro-Benchmarks.** Throughout the implementation process, a key goal was to provide a light-weight mechanism for the LD-DVS algorithm within the real-time scheduler. To that end, we compare the execution costs of this algorithm with that of the single-speed approach employed by Poellabauer et al. (2005).

For the case where LD-DVS must recalculate optimal P-state pairs due to changing cache miss rates or system utilization, we report an execution time of 20-35µs each time the scheduler is invoked, which translates to an execution overhead of 0.20-0.35% relative to the algorithm implemented by Poellabauer et al. (2005). In the case where recalculation of a new optimal P-state pair was not necessary, an execution time of 11-18µs translating to 0.11-0.18% execution overhead was reported. The latter benchmark is nearly identical to the execution overhead of of 0.10-0.19% incurred from the approach presented by Poellabauer et al. (2005).

### 5.1 Memory-Aware DVS

To test the effectiveness of the cache feedback loop, six different programs are run under various service constraints. Of the six test programs, two of the programs chosen are special test programs, designed to generate data cache hits and misses, respectively. The other four programs, madplay, djpeg, cjpeg, and gzip, are chosen to be representative of common memory-bound applications or real-time multimedia applications.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>MAR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcache-hit</td>
<td>0.05</td>
</tr>
<tr>
<td>cjpeg</td>
<td>1.26</td>
</tr>
<tr>
<td>madplay</td>
<td>3.03</td>
</tr>
<tr>
<td>djpeg</td>
<td>3.39</td>
</tr>
<tr>
<td>gzip</td>
<td>4.06</td>
</tr>
<tr>
<td>dcache-miss</td>
<td>11.13</td>
</tr>
</tbody>
</table>

**Table 2. MARs of test programs.**

For the dcache-hit and cjpeg test cases with low MARs (expressed in ‘percentage of instructions executed’ in Table 2), there is little difference between the results of the theoretical frequency scaling and the feedback loop. This is to be expected, since the single scaling factors used by the theoretical algorithm were calculated using a process with a low MAR.

On processes with a higher MAR, such as dcache-miss, gzip, djpeg, and madplay, the performance exhibits the S-shaped curve in Figure 1 as frequency is scaled. In these cases, the feedback loop helps conserve energy at lower utilizations and also helps avoid missed deadlines at higher
utilizations. Figures 8 through 15 show the measured results for dcache-miss, gzip, madplay, and djpeg.

Figures 8, 10, 12, and 14 show the execution times of our test cases. In most cases, the execution times of both the theoretical algorithm and the feedback loop are less than the deadline, indicating that the process met its deadline. For example, in the case of dcache-miss (Figure 8), the feedback loop is able to save more energy in 7 cases, compared to higher energy consumptions in 3 cases, and equal energy consumptions in another 7 cases.

Figures 9, 11, 13, and 15 show the average frequencies of the test applications. In many cases, the feedback loop reduces energy consumption by selecting a lower frequency than the theoretical algorithm. In these cases, a bus speed of 99 MHz is necessary to support the memory references. Since the MAR of the test processes is high, the lowest CPU speed supporting a 99 MHz bus will meet the deadlines in most of the test cases. By detecting the high MAR, the feedback loop selects the lowest CPU frequency supporting the necessary bus speed, resulting in a lower energy consumption over the theoretical algorithm.

However, in a few test cases, the feedback loop is over-aggressive when minimizing clock frequencies, causing it to perform worse than the theoretical algorithm. In other test cases, both the feedback loop and the theoretical algorithm miss their deadlines. In these cases, outside factors such as I/O response time and context switching likely increase the execution time of the process more than expected, resulting in a missed deadline. In further research, we intend to address these challenges, e.g., by implementing a multi-resource feedback algorithm, which measures and takes these factors into consideration when choosing the speeds.

Table 3 summarizes the results of the test cases. In the majority of cases, there is no difference between the execution under the theoretical algorithm and the feedback loop.

This is due to the XScale PXA255’s limited number of frequencies. Since there are only five frequency combinations available, the theoretical algorithm usually picks the optimum frequency. However, particularly at high utilizations and/or high memory access rates, the feedback loop is more accurate in the determination of the optimum frequency.

Out of the 102 test cases, the feedback loop results in fewer missed deadlines in 6% of the cases tested. In another 27% of the cases, the feedback loop results in a lower operating frequency. Overall, these savings lead to an average frequency 8.1% lower than the theoretical DVS algorithm.

Finally, to show that our approach achieves good results when multiple applications compete for the CPU, we ran CPU-bound and memory-bound tasks simultaneously. The results were comparable to the previously shown ones, i.e.,
Table 3. Test processes.

<table>
<thead>
<tr>
<th>Test Program</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
<th>65</th>
<th>70</th>
<th>75</th>
<th>80</th>
<th>85</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcache-miss</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>gzip</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>-</td>
</tr>
<tr>
<td>madplay</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>#</td>
<td>#</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>djpeg</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>#</td>
<td>#</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cjpeg</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>#</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dcache-hit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Legend:

$ Both algorithms meet the deadline, but the feedback loop conserved energy by running at a lower frequency than the theoretical algorithm.

# Both algorithms meet the deadline, but the feedback loop conserved less energy than the theoretical algorithm.

+ The theoretical algorithm misses its deadline, but the feedback loop avoids a missed deadline by raising the frequency.

- Both the theoretical algorithm and feedback loop execute at the same frequency.

Figure 11. CPU/bus frequencies of gzip.

Figure 12. Execution times of madplay.

the feedback-based approach reduces the energy consumption while keeping the number of missed deadlines low. As an example, when executing gzip (CPU-bound) and dcache-miss (memory-bound) simultaneously, the feedback-based approach meets all task deadlines for all CPU utilizations (10%-90%), while the theoretical approach misses the deadline in 22% of the test cases. In addition, the feedback-based approach manages to achieve a lower energy consumption in 44% of the test cases.

5.2 Experiments with LD-DVS

Figure 16 shows the frequency changes of the CPU for a pair of jobs submitted to the Sitsang device. Labels A and B indicate the times when jobs 1 and 2, respectively, begin execution within the system. In both cases the LD-DVS algorithm re-calculates both the optimal E-states. The DVS algorithm measures the progress of a task ($x_{actual}$) over a certain interval and switches to the second frequency whenever $x_{actual}$ exceeds the computed $x$ value of the corresponding E-state. The re-computation of E-states at labels C and D is caused by a significant change in the workload (i.e., cache miss rates) of the tasks.

To illustrate the energy savings that are achievable by our approach, consider the energy consumed by the different regions marked in Figure 16. The energy consumed during execution in the region marked B is found to be lower by 8.89% than the energy consumed during execution in the region D. This is due to the fact that during region B frequencies were chosen that were lower than the set of fre-
queencies in region D. The significant increase in the MAR led to a transition to a region with higher frequencies to accommodate the increased workload. A similar scenario can be observed in Figure 17 where an energy difference of 9.13% was reported between regions B and D. Both the calculations were made over a uniform range of execution times. The optimal frequency pair in our approach is always selected in accordance with the current workload of the system.

5.3 Reducing Switching Overheads

We further considered the need for additional techniques to fine-tune the performance of the algorithm. For example, the performance of the algorithm can be improved if the frequency selection can be made more stable and the number of frequency transitions can be reduced. Particularly the latter can significantly reduce the overheads. This can be achieved by introducing a hysteresis factor into the algorithm, i.e., whenever \( x_{\text{actual}} \) is compared to E-state ratios, the hysteresis factor can reduce the number of transitions. This modification to Algorithm 1 would affect lines 20 and 25, where \( x_{\text{actual}} \) would be compared to \((x + HysteresisFactor)\) and \((100 - x + HysteresisFactor)\), respectively. This region of tolerance implies that the frequency is switched between the computed pair of frequency values only when the proportion of time spent at one frequency level exceeds the tolerance region boundary. Different ‘dead zone’ regions were achieved by varying the hysteresis factors which were chosen heuristically. Figure 17 shows the actual run-time CPU frequency profile for a real-time application scheduled with the LD-DVS algorithm with 10% hysteresis. The presence of a hysteresis factor leads to a marked reduction in the number of P-state transitions required during execution. At the same time, the
reduction in the number of transitions hinders the ability of the algorithm to maximize energy savings in the system. However, this trade-off may still be useful in order to ensure that the tasks’ real-time deadlines are met which are otherwise affected by the latency involved in switching between P-states. Our future work will further study the online computation of optimal hysteresis values (i.e., a value that maintains the responsiveness of the algorithm while minimizing the number of frequency transitions).

6 CONCLUSIONS

This paper discussed the integration of two recent techniques used for DVS algorithms. First, system-wide energy management requires that a DVS algorithm is aware of other resources in a system, most notably, I/O resources. Considering these resources and their effects on task runtimes allows for more accurate predictions of future system utilization and thereby frequency computations. Second, dual-speed DVS has been introduced as a technique to approximate the optimal frequency of the system, when only a small number of speed levels are supported.

The introduced algorithm, LD-DVS, combines both techniques to achieve lower energy consumptions. First, utilizing the processor’s performance counters, LD-DVS monitors memory access rates (or cache misses) and better predicts future system utilization or task run-time, resulting in more precise voltage and frequency computations. Second, our experimental studies show that selecting the two neighboring discrete speed levels (in order to approximate an optimal theoretical speed) does not always yield the highest energy savings. Therefore, LD-DVS computes a pair for frequencies that also approximate the optimal speed, but minimize the energy consumption. Our studies also show that the selection of a frequency pair depends on the current workload (e.g., memory- versus CPU-intensive tasks). Thus, LD-DVS recomputes the frequencies whenever task sets change or the memory access rate of a task changes.

Our future work will extend the efforts presented in this paper to include other device components capable of energy management, most importantly I/O devices such as storage or networks, which typically offer low-power idle or sleep modes. Also, we introduced the notion of hysteresis in switching between two frequency levels. As a future direction, we will explore the use of an adaptive hysteresis mechanism that will receive feedback from the processor about the current number of frequency transitions and tune itself to an optimum value in accordance with the current status of the system. Such an approach will help reduce the overheads involved in switching between frequencies and at the same time help achieve maximum energy savings. Finally, we will explore more thoroughly the effects the introduced DVS algorithm has on overall system-wide energy consumption, especially when considering feedback from several I/O resources.

ACKNOWLEDGMENT

The authors wish to thank Leo Singleton who contributed to our early work on DVS on the Sitsang platform. This research was supported in part by NSF grant CNS-0545899 and by Intel Corporation.

References


