Monitoring of Cache Miss Rates for Accurate Dynamic Voltage and Frequency Scaling

Leo Singleton[†], Christian Poellabauer[††], Karsten Schwan[†]
[†]College of Computing, Georgia Institute of Technology
[††]Department of Computer Science & Engineering, University of Notre Dame

Abstract

Modern mobile processors offer dynamic voltage and frequency scaling, which can be used to reduce the energy requirements of embedded and real-time applications by exploiting idle CPU resources, while still maintaining all applications' real-time characteristics. However, accurate predictions of task run-times are key to computing the frequencies and voltages that ensure that all tasks' real-time constraints are met. Past work has used feedback-based approaches, where applications' past CPU utilizations are used to predict future CPU requirements. Inaccurate predictions in these approaches can lead to missed deadlines, less than expected energy savings, or large overheads due to frequent voltage and frequency changes. Previous solutions ignore other 'indicators' of future CPU requirements, such as the frequency of I/O operations, memory accesses, or interrupts. This paper addresses this shortcoming for memory-intensive applications, where measured task run-times and cache miss rates are used as feedback for accurate run-time predictions. Cache miss rates indicate the frequency of memory accesses and enable us to derive the latencies introduced by these operations. The results shown in this paper indicate improvements in the number of deadlines met and the amount of energy saved.

1 Introduction

Background. The increasing energy demands of more powerful mobile processors can lead to shorter battery lives and greater heat dissipation. In order to reduce these demands while preserving high performance, energy-awareness has become a critical factor in the design of mobile and embedded systems. As a result, new methods for dynamic energy management have been introduced, e.g., dynamic voltage and frequency scaling (DVFS). DVFS takes advantage of the quadratic relationship between supply voltage and energy consumption [1], which can result in significant energy savings. However, DVFS techniques pose a difficult challenge to real-time systems: not only must DVFS be used to reduce energy consumption, but it must do so without impacting the desired quality of service offered by the system to applications and end users. Recent approaches to solving this issue bases the frequency/voltage computations used by the scheduler on its predictions of future task run-times [8, 4]. However, these approaches ignore that applications' run-times also depend on I/O, memory accesses, and interrupts, resulting in inaccuracies in run-time predictions.

This paper addresses inaccuracies in run-time predictions by removing one of their major causes, the utilization of memory resources. Memory-intensive applications, including image and video processing and scientific applications, require frequent accesses to memory, which contributes substantially to their total run-times, particularly in systems where DVFS not only changes the CPU's clock frequency, but also the frequency of the bus linking CPU and memory. Utilizing all tasks' scheduling attributes, a DVFS algorithm can compute the total CPU utilization, and consequently, the frequency and voltage combination that maximizes this utilization (i.e., 100%). However, inaccuracies in predicting tasks' run-times can cause system utilization to exceed 100%, thereby leading to missed deadlines and frequent frequency and voltage re-computations. Overestimating the utilization can result in inefficient energy management. The key idea explored in this paper, therefore, is to dynamically monitor a task's cache miss rate, which indicate the memory overheads experienced by the task. These miss rates are used as inputs by the DVFS algorithm used to better predict run-time and more accurately compute the frequencies and voltages to be used for task execution. Experimental results utilize the frequency scaling capabilities of an XScale-based evaluation board. Future work will extend these results to voltage scaling.

Related Work. Previous work on DVFS algorithms for real-time schedulers described in [4] and [5] relies on worst-case task execution times, whereas we use cache feedback data to approximate actual execution times. This is a first

step towards more accurate run-time predictions and optimal energy savings; our future work will address further causes for inaccuracies, e.g., disk and network I/O delays. There has also been substantial prior work that examines the effects of DVFS on cache performance. In [3], the authors evaluate energy savings due to the reduced cache miss penalties at lower clock frequencies. Compile-time techniques, such as the one proposed in [2] have exploited these stall periods. Online algorithms including [7] utilize performance counters available in modern microprocessors to achieve similar energy savings without the need for recompiling applications.

2 Dynamic Voltage and Frequency Scaling

Real-Time CPU Scheduling. To efficiently support real-time applications, each task i has a period T_i associated, where the end of the current period is the deadline. Each task i is guaranteed C_i time units 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). If a task misses its deadline, the scheduler violated the real-time guarantees to this task.

Clock frequencies are typically computed such that the *rest utilization* of the CPU is exploited by slowing down task execution. We compute a new clock frequency whenever the system utilization changes, e.g., when tasks join or leave the run queue. The current utilization of all tasks is computed with:

$$U = \sum \frac{C_i}{T_i} \tag{1}$$

C_i is the service time allocated to task i at the default clock frequency and this service time increases when the clock is slowed down. For each clock frequency n, a *scaling factor* k_n can be obtained by executing a sample processing-intensive code at both the default frequency f_{max} and f_n and dividing the measured run-times: $k_n = C_n/C_{max}$. This is repeated for each available clock frequency (or core voltage) for a given processor. The goal of frequency scaling is to get as close to 100% utilization as possible, i.e.,

$$U_{100\%} = \sum \frac{C_i * k'}{T_i} \tag{2}$$

where k' is the yet unknown scaling factor. To guarantee that best-effort tasks are not starved, we can replace $U_{100\%}$ with $U_{x\%}$, e.g., $U_{95\%}$. Then k' can be determined with:

$$k' = \frac{U_{95\%}}{\sum \frac{C_i}{T_i}} \tag{3}$$

The resulting k' is compared to the previously obtained scaling factors, and the scaling factor k_n closest to k' ($k_n \leq k'$) is selected, and the clock frequency is adjusted to frequency n.

Note that other approaches exist that compute different scaling factors for each individual application, however, these approaches have higher computational overheads. Nevertheless, the solution introduced in this paper could also be applied to these approaches. However, this common approach to DVFS assumes that the run-times can be predicted by monitoring previous CPU requirements, ignoring that other factors (e.g., I/O utilization) can affect the run-times, making the predictions inaccurate. This issue is addressed by the remainder of this paper.

Run-Time Prediction. Intuitively, the scaling factor k' would be the ratio of the clock frequencies. However, various factors cause a change in k'. Figure 1 shows the performance of three applications, relative to the maximum frequency. On the x-axis, we display the different frequencies supported by our 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. Note however, that 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 dcache-miss, 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. If a simple linear model were used to determine the k' factors, it would underestimate the utilization at lower

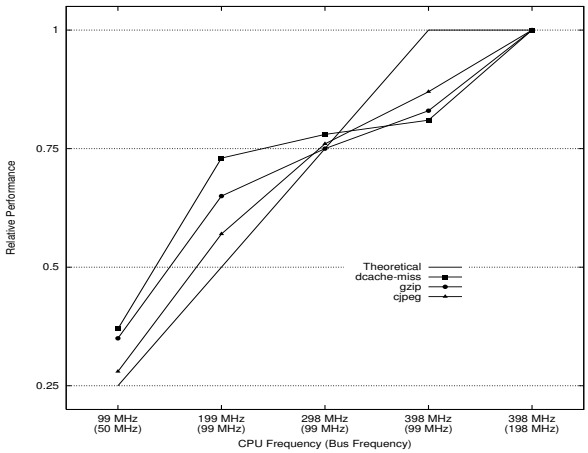

Figure 1: Relative performance of three applications at various clock frequencies on the Intel XScale PXA255.

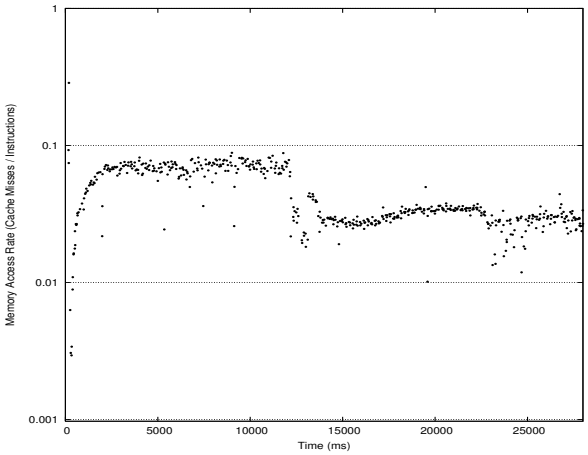

Figure 2: Data memory access rate of gzip.

clock frequencies, resulting in missed deadlines, and overestimate the utilization at higher clock frequencies, resulting in wasted energy. The results in Figure 1 point out the importance of considering cache miss rates in the DVFS approach.

The remainder of this paper will use the *Memory Access Rate (MAR)* to quantify the rate of cache misses. As opposed to the *miss rate*, MAR is defined in Equation 4 as the ratio of data cache misses to instructions executed.

$$MAR = \frac{data\ cache\ misses}{instructions\ executed} \tag{4}$$

Figure 2 shows gzip's MAR plotted over time. Applications like gzip experience a large variation in MAR, with the rate changing by a factor of over a hundred. Due to this variation, a single set of k' values would not be sufficient for the entire execution of gzip. Therefore, it is critical to determine the appropriate set of k' factors for each process's current behavior.

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 [6]. Therefore, we use MAR as indicator for the performance penalties incurred by memory accesses.

3 Cache Miss Rates as Feedback for DVFS

Approach. To improve run-time predictions for DVFS, we propose to monitor cache miss rates using a software feedback loop. Our approach uses performance counters to count the number of data cache misses and instructions executed, and keep a weighted average of the MAR for each process. From this value, the CPU scheduler can pick an appropriate matrix of scaling factors, which will more accurately estimate the process's execution time under the available frequencies.

Our approach is implemented on an Intel Sitsang evaluation board with an XScale PXA255 processor. The PXA255 processor has three caches, a 32KB instruction cache (32-way associativity), a 2KB L1 mini-data cache (2-way associativity), and a 32KB L2 data cache (32-way associativity). All caches use the round-robin policy.

Performance Counters. Many processors support some form of performance counters for optimization. These hardware counters support the monitoring of numerous types of events, such as memory accesses and pipeline stalls. The Intel XScale PXA255 processor has three 32-bit performance counters: one which counts clock cycles and two general-purpose counters, which may be used to monitor any of 16 possible events, selectable in software via a configuration register[1].

For these experiments, it is important to choose performance counters which are not affected by frequency changes. Since the execution time varies with frequency, cache misses over clock cycles would vary for a given program depending

[1] http://www.intel.com/design/pca/prodbref/252780docs.htm

on the frequency. Therefore, for the computation of MAR, instructions executed was chosen instead of clock cycles. Unfortunately, the PXA255 does not provide a performance counter for combined cache misses, only separate ones for the instruction and data caches. Due to the limitations of the processor, data cache misses was chosen. Hence, the definition of the MAR as the ratio of data cache misses to instructions executed (Equation 4).

Monitoring of Cache Miss Rates. To implement an energy-aware scheduler using cache feedback, we first must approximate the execution time of a process with a given MAR. If we take a program's execution time, and break it into two components, time spent executing instructions and time spent stalled on memory accesses, we get the following formula:

$$t_{execution} = C_{CPU}(f_{CPU}) + (MAR * C_{bus}(f_{bus})) \tag{5}$$

where $C_{CPU}(f_{CPU})$ is a constant dependent on CPU clock frequency, and $C_{bus}(f_{bus})$ is a constant dependent on bus frequency. To find the values for these two constants, a test program was designed to generate a specified miss rate by performing memory accesses on a large array. The test program was run with multiple MARs at each supported frequency, and using linear regression, the execution times were used to determine the constants C_{CPU} and C_{bus}. The experimental determination of the C_{bus} constants ensures that processor techniques to hide memory latencies are considered, i.e., C_{bus} only indicates the memory latencies that could not be hidden by the processor.

The scaling factors for each frequency can be calculated using the ratio of the execution times from the equation above. Calculating these scaling factors at runtime would add extra overhead to each scheduler invocation, so twelve matrices of scaling factors were precalculated and compiled into the DVFS module. The twelve matrices were selected such that matrix M_n corresponds to the scaling factors for processes with an MAR of 2^{-n}. A field was added to Linux's task_struct to keep a weighted average of the MAR for each process, and code was added to select the appropriate matrix of k' factors at each scheduler invocation.

4 Experimental Evaluation

To test the effectiveness of the cache feedback loop, four different programs are run under various service constraints. Two of the programs chosen are test cases designed to generate data cache hits and misses. The other two programs, cjpeg, and gzip, are chosen to be representative of common memory-bound or real-time multimedia applications. For the dcache-hit test case, with the lowest MAR, there is little difference between the results of the theoretical frequency scaling and the feedback loop. This is to be expected, since the theoretical algorithm does not consider memory accesses. On processes with a higher MAR, such as dcache-miss and gzip, the performance exhibits the S-shaped curve in Figure 1. In these cases, the feedback loop helps conserve energy at lower utilizations and also helps avoid missed deadlines at higher utilizations.

Table 1 summarizes the 68 test cases; detailed results have been omitted for brevity.

	Utilization (%)																
Test Program	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
dcache-miss	-	-	-	$	$	-	-	-	$	$	$	$	$	+	-	+	-
gzip	-	-	-	$	$	-	-	-	$	$	$	-	-	-	#	#	-
cjpeg	-	-	-	$	$	-	-	-	$	$	$	-	-	-	-	#	-
dcache-hit	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

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.

Table 1: Test processes.

In the majority of cases, there is no difference between the execution under the theoretical algorithm and the feedback loop, due to the PXA255's limited number of frequencies. Since there are only five frequency combinations available, the theoretical algorithm usually picks the optimum frequency. In many cases, the feedback loop reduces energy consumption by selecting a lower frequency than the theoretical algorithm. In these cases, the bus speed is usually the limiting factor for performance. Since multiple CPU frequencies share the same bus speed, the feedback loop can select the lowest CPU frequency, 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 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 implement a multi-resource feedback algorithm, measuring and taking these factors into consideration when choosing optimal frequencies.

Overall, the feedback loop results in fewer missed deadlines in 3% of the cases tested. In another 25% of the cases, the feedback loop results in a lower operating frequency. This lead to an average frequency savings of 8.1%.

5 Summary and Conclusion

Dynamic voltage and frequency scaling allows for a reduction in energy consumption and in heat dissipation for mobile devices. However, most DVFS algorithms are not well-suited to real-time applications, since they will either miss deadlines or waste energy if the execution time of the process is miscalculated. Utilizing the processor's performance counters to monitor memory accesses, we propose to better predict process execution time using a feedback loop. Here, the DVFS approach considers the recent number of cache misses, deriving the overheads for accessing memory, and thereby better predicting an application's run-time, resulting in more precise voltage and frequency computations.

When tested with four applications under various utilizations, the cache feedback method results in an energy savings in 25% of the cases tested. At high utilizations and high memory access rates, the feedback loop raised the CPU frequency, avoiding missed deadlines in 3% of the cases tested.

In future research, we plan to monitor additional resources to use in our feedback loop, including I/O requests and context switching overheads. While the approach introduced in this paper relies on frequency scaling, the XScale-based evaluation board used in our experiments also supports voltage scaling. Therefore, our future results will leverage both frequency and voltage scaling to achieve optimal energy management results.

References

[1] T. Burd and R. Brodersen. Energy Efficient CMOS Microprocessor Design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297, January 1995.

[2] C. Hsu and U. Kremer. Compiler-Directed Dynamic Voltage Scaling for Memory-Bound Applications. In *Proceedings of the Workshop on Power-Aware Computer Systems, in conjunction with ASPLOS-IX*, November 2000.

[3] D. Marculescu. On the Use of Microarchitecture-Driven Dynamic Voltage Scaling. In *Proceedings of the Workshop on Complexity-Efective Design, in conjunction with ISCA-27*, June 2000.

[4] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems. In *Proc. of the 18th Symposium on Operating Systems Principles*, October 2001.

[5] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2003.

[6] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided Region Prefetching: A Cooperative Hardware/Software Approach. *ACM SIGARCH Computer Architecture News*, 2003.

[7] A. Weissel and F. Bellosa. Process Cruise Control. In *Proc. Compilers, Architectures and Synthesis for Embedded Systems*, pages 238–246, October 2002.

[8] Y. Zhu and F. Mueller. Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2004.