# Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-off based on the Ratio of Off-chip Access to On-chip Computation Times[*]

Kihwan Choi, Ramakrishna Soma, and Massoud Pedram
Department of EE-Systems, University of Southern California, Los Angeles, CA90089
{kihwanch, rsoma, pedram}@usc.edu

***Abstract***
*This paper presents an intra-process dynamic voltage and frequency scaling (DVFS) technique targeted toward non real-time applications running on an embedded system platform. The key idea is to make use of runtime information about the external memory access statistics in order to perform CPU voltage and frequency scaling with the goal of minimizing the energy consumption while translucently controlling the performance penalty. The proposed DVFS technique relies on dynamically-constructed regression models that allow the CPU to calculate the expected workload and slack time for the next time slot, and thus, adjust its voltage and frequency in order to save energy while meeting soft timing constraints. This is in turn achieved by estimating and exploiting the ratio of the total off-chip access time to the total on-chip computation time. The proposed technique has been implemented on an XScale-based embedded system platform and actual energy savings have been calculated by current measurements in hardware. For memory-bound programs, a CPU energy saving of more than 70% with a performance degradation of 12% was achieved. For CPU-bound programs, 15~60% CPU energy saving was achieved at the cost of 5-20% performance penalty.*

## 1 Introduction

Demand for low power consumption in battery-powered computer systems has risen sharply. This is because extending the service lifetime of these systems by reducing their power requirements is a key customer/user requirement. More recently, low power design has become a critical design consideration even in high-end computer systems, due to expensive cooling and packaging costs and lower reliability often associated with high levels of on-chip power dissipation.

Dynamic voltage and frequency scaling (DVFS) technique has proven to be a highly effective method of achieving low power consumption while meeting the performance requirements [1]. The key idea behind DVFS techniques is to dynamically scale the supply voltage level of the CPU so as to provide "just-enough" circuit speed to process the system workload while meeting the total compute time and/or throughput constraints, and thereby, reducing the energy dissipation (which is quadratically dependent on the supply voltage level.) A number of modern microprocessors such as Intel's XScale [2] and Transmeta's Cruso [3] are equipped with the DVFS functionality.

DVFS techniques may be used to reduce the energy consumption of an executed task while ensuring that the task meets its deadline. However, these techniques are not directly applicable to general-purpose operating systems because they assume that critical information about all tasks, such as the task arrival time, deadline, and workload, are known in advance. Moreover, the workload of a task is often represented by the number of CPU clock cycles required to complete the task regardless of whether the workload consists of mainly CPU-bound or memory-bound instructions. The latter information is, of course, critical in determining the idle time of the CPU.

We are interested in a DVFS policy for general-purpose computer systems that differentiates between CPU-bound and memory-bound instructions in the workload. The intuition for workload partitioning is as follows. Memory is asynchronous with the processor and often has its own clock. Now if the task execution time is dominated by the memory access time, then the CPU speed can be slowed down with little impact on the total execution time. This could, however, result in potentially significant savings in energy consumption.

In this paper, we propose an intra-process DVFS technique for non real-time operation in which finely tunable energy and performance trade-off can be achieved. The main idea is to lower the CPU frequency during the CPU idle times, which are, in turn, due to external memory stalls. To capture the CPU idle time at run time, several performance monitoring events, provided by performance monitoring unit (PMU) in the XScale processor, are used. The proposed technique has been implemented on an embedded system platform and actual energy savings have been calculated by current measurements in hardware. On this platform more than 70% CPU energy savings was achieved for memory-bound programs with a performance degradation of only 12%. In contrast, 15~60% CPU energy savings was achieved for CPU-bound programs with a performance degradation of 5-20%.

The main contributions of our work are: (1) It presents one of the first actual implementations of an intra-process DVFS policy that exploits dynamic events at run time without any support from compiler or modification of the application program itself. (2) A simple, but effective, regression model is proposed to approximately determine the CPU idle time due to memory stalls by estimating the ratio of the total off-chip access time to the total on-chip computation time at runtime. (3) Evaluation of the proposed method is performed through actual hardware measurements for a number of different applications.

The remainder of this paper is organized as follows. Related work is described in Section 2. In Section 3 and 4, a new DVFS policy is presented. Details of the implementation, including both

[*] This research was supported in part by DARPA PAC/C program under contract DAAB07-02-C-P302 and by NSF under grant no. 9988441.

hardware and software, are described in Section 5. Experimental results and conclusions are given in Sections 6 and 7, respectively.

## 2 Prior Work

Previous DVFS-related works may be divided into two categories based on the scaling granularity: coarse-grained and fine-grained. Coarse-grained voltage scaling is performed at the operating system (OS) or application level, whereas fine-grained voltage scaling is performed at the level of individual blocks/segments in an application task or software program. Many scheduling policies for hard real-time applications have coarse granularity. Multi-task scheduling in the OS is the focus of [4][5][6][7]. More precisely, scheduling is performed at task level by the OS so as to reduce energy consumption while meeting hard timing constraints for each task. In these coarse-grained DVFS approaches, it is assumed that the total number of CPU cycles needed to complete each task is fixed and known a priori. There are also a number of studies that implement fine-grained DVFS as part of compile-time optimization or by modifying the application program itself. In [8], an intra-task voltage scheduling technique was proposed in which the application code is divided into many segments and the worst-case execution time of each segment (which is obtained from a static timing analysis) is used to determine a suitable voltage for the next segment. In [9] a method based on a software feedback loop was proposed. In this method, a deadline for each time slot is provided. The authors calculate the operating frequency of the processor for the next time slot depending on the slack time generated in the current slot and the worst-case execution time of the next time slot. In [10], a checkpoint-based algorithm is proposed in which the scaling points are identified off-line by the compiler. In [11] and [12], compiler-assisted DVFS techniques were proposed, in which frequency is lowered in memory-bound region of a program with little performance degradation.

DVFS approaches that rely on micro-architecture or embedded hardware without any assistance from a compiler or a simulator have been reported. In [13] a microarchitecture-driven DVFS technique was proposed in which cache miss drives the voltage scaling. In [14] IPC (instruction per cycle) rate of a program execution was used to direct the voltage scaling. Reference [15] used a performance monitoring unit (PMU) to produce the optimal frequency and voltage levels under a given performance degradation. The PMU captures the dynamic program behavior such as cache hit/miss ratio and memory access counts during the whole execution time.

A heuristic technique was proposed in [11] in which voltage scaling is done by identifying memory-bound regions of a program trace. However, this work needs compiler support to identify such regions. There is a different voltage scaling approach, called process cruise control, where dynamic events from the PMU on an XScale processor are used to determine the optimal frequency for a performance loss constraint [15]. In particular, the authors defined optimal frequency domains in 2-D memory vs. instruction count space. This approach requires no help from off-line simulation or compiler and only relies on dynamic event counts from the PMU. However, it is not flexible in the sense that frequency domains are obtained through extensive experiments of micro-benchmarks for a given performance loss (set to 10% in that work) and this performance loss is fixed for all different applications. This stiff policy does not allow a precise and graceful control of the energy-performance trade-off.

In this paper, we propose a DVFS policy for non real-time application similar to the one presented by [15]. However, in our proposed DVFS approach, we use the performance events in a different way. Furthermore, our policy enables more precise control over energy-performance trade-off by using regression-based method in which performance events are used to recognize memory-bound region at runtime effectively.

## 3 Performance-energy trade-offs

### 3.1 Performance degradation and energy saving

To perform ideal DVFS, we have to accurately predict the execution time of a task at any clock frequency. The execution time is a function of the instruction mix (the sequence of unrolled instructions to be executed) and the cycle-per-instruction (CPI). A RISC instruction mix consists of register-type instructions, memory-type instructions, and branch-type instructions (the control instructions for supervisor mode are not considered here). After the application is compiled from the source code into the object code, the ratios between these three instruction-types in the instruction mix will become fixed if the control flow is known at compile time. The CPI of the instruction mix depends on not only the instruction-types and the data dependency, but also the run-time factors such as SDRAM access latency, PCI access latency, other running processes, etc.

The instruction latencies can be classified as on-chip latencies (data dependency, TLB hits, cache hits, branch prediction) or off-chip latencies (memory latency due to cache misses, PCI latency due to access to the frame buffer). The on-chip latencies are caused by events that occur inside the CPU (e.g., data dependency). They are synchronized to the internal clock and may linearly be reduced by increasing the CPU frequency. The off-chip latencies (e.g. the SDRAM and PCI latencies), on the other hand, are independent of the internal frequency and are thus not affected by changing the CPU frequency. Accesses to external devices such as SDRAM, PCMCIA flash card, LCD display, and USB storage are synchronized to the bus clock, which is independent of the CPU frequency.

Let $T$, $T_{onchip}$, and $T_{offchip}$ denote the total execution time of a program, the on-chip computation time, and the off-chip access time, respectively. T is obviously written as:

$$T = T_{onchip} + T_{offchip} \qquad (1)$$

Notice that this breakdown of the total execution is not exact when the target processor supports out-of-order execution whereby instructions after the instruction that caused an off-chip access may be executed during the off-chip access. In such a case, $T_{onchip}$ and $T_{offchip}$ can overlap. However, in practice, the error introduced in this way is quite small considering that the memory access time is about two orders of magnitude greater than the instruction execution time. Therefore, out-of-order execution does not cause a large error in equation (1).

When the CPU frequency changes, the change in $T$ is solely due to $T_{onchip}$:

$$\frac{\Delta T}{\Delta f} = \frac{\Delta T_{onchip}}{\Delta f} \quad , \quad \frac{\Delta T_{offchip}}{\Delta f} \approx 0 \qquad (2)$$

The increased execution time of a program due to lowered clock frequency represents the performance loss ($PF_{loss}$), which is defined as follows:

$$PF_{loss} = \frac{(T_{f_n} - T_{f_{\max}})}{T_{f_{\max}}} \qquad (3)$$

where $f_{max}$ is the maximum frequency of the CPU, $f_n$ is a frequency lower than $f_{max}$, $T_{fn}$ and $T_{fmax}$ are the total task execution times at CPU frequencies of $f_n$ and $f_{max}$, respectively.

For a given program, different ratios of $T_{onchip}$ and $T_{offchip}$ result in very different $PF_{loss}$ over CPU frequencies. Figure 1 provides energy-performance trade-offs for various applications. For example, in case of the "crc" and "djpeg", lowering frequency introduces significant performance loss compared to other tasks implying that these programs are CPU-bound (i.e., $T_{onchip}$ >> $T_{offchip}$). On the contrary, it is known that "fgrep" and "qsort" are *memory-bound* (i.e., $T_{onchip} << T_{offchip}$) by observing little performance degradation with lowered frequency. Based on these observations, we conclude that the ratio of $T_{onchip}$ to $T_{offchip}$ for a program is very important to the degree of energy saving and performance penalty attained by DVFS techniques.

In general, the execution time of a program can be represented in terms of the CPI, the number of instructions being executed, and the CPU frequency [16]. More precisely, $T_{onchip}$ and $T_{offchip}$ can be represented as follows:

$$T_{onchip} = \frac{\sum_{i=1}^{n} CPI^{i}_{onchip}}{f_{cpu}} = \frac{n \cdot CPI^{avg}_{onchip}}{f_{cpu}} \qquad T_{offchip} = \frac{\sum_{j=1}^{m} CPI^{j}_{offchip}}{f_{mem}} = T - T_{onchip} \tag{4}$$

where $n$ is the total number of instructions in the instruction stream, $m$ is the number of off-chip accesses in that stream, $CPI^{i}_{onchip}$ denotes the number of CPU clock cycles for the ith instruction, $CPI^{j}_{offchip}$ denotes the number of memory clock cycles for the jth off-chip access, $CPI^{avg}_{onchip}$ denotes the average on-chip CPI, $f_{cpu}$ and $f_{mem}$ denote the *current* clock frequency of the CPU and the clock frequency of the off-chip bus. It should be pointed out that $f_{mem}$ can assume different values depending on the external devices being accessed. For example, in our test system, 100MHz clock frequency is used for the SDRAM access whereas 33MHz speed is used for the PCI-peripheral devices. Note that $f_{mem}$ cannot be scaled.

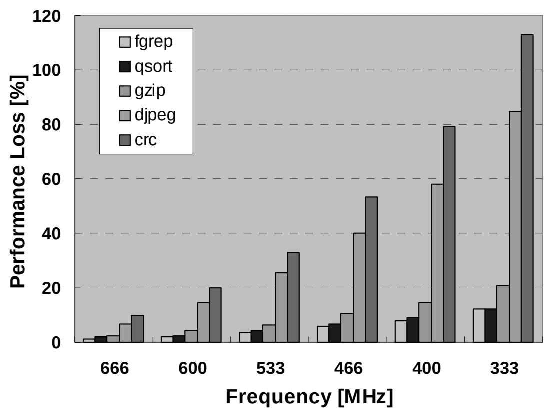


**Figure 1: Performance loss changes according to CPU frequency.**

**Definition 1:** The $\beta$ value of a program is defined as the ratio $T_{offchip}/T_{onchip}$ for that program.

$\beta$ represents the degree of potential energy saving because the larger $\beta$ is, the more CPU energy saving can be achieved by a DVFS technique. Consequently, we need accurate information about $\beta$ in order to sustain an effective DVFS technique.

From equations (3) and (4), the optimal frequency, $f_{target}$, for a given $PF_{loss}$ value is calculated as follows:

$$f_{target} = \frac{f_{\max}}{1 + PF_{loss} \cdot \left[1 + \beta \cdot \left(\frac{f_{\max}}{f_{cpu}}\right)\right]} \tag{5}$$

As it can be seen from the above equation, $f_{target}$ is closely related to $\beta$ of a program. Consequently, accurate calculation of $\beta$ is quite important to the effectiveness of our proposed DVFS approach.

## 3.2 Scaling granularity

The ideal DVFS can instantaneously change the voltage/frequency values. In reality, however, it takes time to change CPU frequency/voltage due to factors such as the internal PLL (phase lock loop) locking time and capacitances that exist in the voltage path. For the 80200 XScale processor, the latency for switching the CPU voltage/frequency is 6 μsec at 333MHz [2]. The minimum quantum of time for scaling the CPU frequency/voltage must be at least two to three orders of magnitude larger than this switching latency. At the same time, we would like to minimize the overhead of the voltage/frequency scaling as far as the OS is concerned. Therefore, we use the start time of an (OS) *quantum* (approximately 50msec in Linux) used by the OS to schedule processes as DVFS decision points, that is, each time the OS invokes the scheduler to schedule processes in the next quantum, we also make to decision as to whether or not the CPU voltage/frequency is changed and if so, scale the voltage/frequency of the CPU.

## 3.3 Events monitored through the PMU on XScale

It is very difficult to calculate the exact $\beta$ of a program in a static manner such as during the compilation time because on/off-chip latencies are severely affected by dynamic behavior such as cache statistics and different access overheads for different external devices. So, these unpredictable dynamic behaviors should be captured at run time. This can be achieved by using a performance-monitoring unit that is often available in modern microprocessors. In our target system, the CPU is Intel's XScale, which supports monitoring of 20 performance events including cache hit/miss, TLB hit/miss, and number of executed instructions. The overhead for accessing PMU (read/write) is less than 1usec [15] and can be ignored. However, there is a limitation in using these events in the sense that only two events can be monitored at the same time along with the number of clock counts in a quantum (CCNT).

For our DVFS policy, we performed many experiments to figure out which events can give valuable clue about $\beta$ and the following two events were proven to be most helpful based on experimental results: (i) the number of instructions being executed (INSTR) and (ii) the number of memory accesses (MEM).

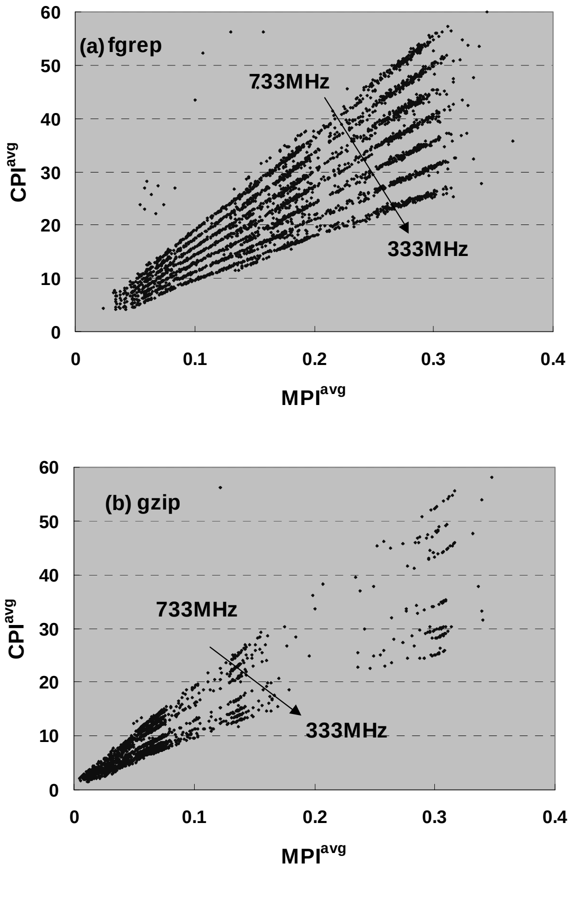


**Figure 2: Contour plots of $CPI^{avg}$ versus $MPI^{avg}$ for different CPU clock frequencies**

Using these two events, INSTR and MEM, along with CCNT, $CPI_{onchip}$ can be extracted as in Figure 2. Figure 2 plots the combination of three events while executing (a) "fgrep" and (b) "gzip" applications at different frequencies from 733MHz to 333MHz at a fixed step of 66MHz. At the start of each quantum, the PMU reports the CCNT, INSTR, and MEM. From these three parameter values, we can calculate the average CPU cycles per instruction ($CPI^{avg}$) for the instruction stream as the ratio of CCNT to INSTR. Similarly, we can calculate the average memory cycles per instruction ($MPI^{avg}$). In this figure, we have plotted $CPI^{avg}$ on the y-axis and $MPI^{avg}$ on the x-axis. Each dot in the plot represents one PMU report. From this figure, we can easily see that, at a fixed CPU clock frequency, $CPI^{avg}$ is linearly related to $MPI^{avg}$ as follows:

$$CPI^{avg} = b(f) \cdot MPI^{avg} + c \tag{6}$$

where $b(f)$ is frequency-dependent slope. Notice that intercept $c$ is equal to the average on-chip CPI, $CPI^{avg}_{onchip}$ and is independent of frequency $f$. Therefore, Eq. (6) can be used to provide an accurate estimation of $CPI^{avg}_{onchip}$ from which $\beta$ can be determined from Eq. (4) and Definition 1.

## 4 Regression-based Fine-Grained DVFS

### 4.1 Calculating $\beta$ with a regression equation

In our proposed DVFS approach, monitored event values are used to estimate coefficient $b$ and $c$ of regression Eq. (6), and then to use this equation to predict $\beta$ of a program. Voltage/frequency scaling is performed at the start of each quantum. Regression coefficients $b$ and $c$ are dynamically updated as explained below.

Let the linear equation for the regression be $y=b*x+c$, where $x$ and $y$ denote $MPI^{avg}$ and $CPI^{avg}$, respectively. Coefficients $b$ and $c$ at quantum $t \geq N$, are calculated from the last $N$ PMU reports as follows:

$$b = \frac{N \cdot (\sum_{i=t}^{t-N+1} x_i \cdot y_i) - (\sum_{i=t}^{t-N+1} x_i) \cdot (\sum_{i=t}^{t-N+1} y_i)}{N \cdot (\sum_{i=t}^{t-N+1} x_i^2) - (\sum_{i=1}^{t-N+1} x_i)^2}, \quad c = \frac{\sum_{i=t}^{t-N+1} y_i}{N} - b \cdot \frac{\sum_{i=t}^{t-N+1} x_i}{N} \tag{7}$$

where $x_i$ and $y_i$ denote the $MPI^{avg}$ and $CPI^{avg}$ for the $i^{\text{th}}$ quantum. Note that we must choose $N$ carefully since if $N$ is chosen to be too small, we will be too sensitive to small changes in the program behavior and we may not have enough data points to do a good regression. On the other hand, if $N$ is too large, then we may potentially filter out many important changes in the program behavior. The regression coefficients are updated at the start of every quantum. Recall that the regression equation is maintained for each frequency because $b$ is different for different frequencies.

The optimal frequency for the next quantum $t+1$ is calculated as follows. After quantum $t$, $\beta$ of quantum $t$, $\beta^t$, is calculated as:

$$\beta^t = \frac{CPI^{avg,t}}{CPI^{avg,t}_{onchip}} - 1 \tag{8}$$

Once $\beta^t$ is obtained, the target CPU frequency for the next quantum, $f^{t+1}$, is calculated from Eq. (5) with the specified $PF_{loss}$ as follows:

$$f^{t+1} = \frac{f_{\max}}{1 + PF_{loss} \cdot \left[1 + \beta^t \cdot \left(\frac{f_{\max}}{f^t}\right)\right]} \tag{9}$$

### 4.2 Prediction error adjustment

We assumed that $\beta$ of the next quantum is the same as that of the current quantum. However, in reality, $\beta$ varies in different quanta during the program execution. This is due to different off-chip latencies for the SDRAM and PCI-device accesses. Furthermore, different applications have different $\beta$ distributions during runtime. This situation becomes worse when the quantum length is varied, for example, when a process performs an I/O operation (mostly file-read/write operations). In such a case, the CPU preempts the process, thus, the length of the quantum is shortened compared to the "standard" quantum length of approximately 50msec.

To alleviate these shortcomings, we modify the proposed technique in order to handle the non-equal quantum times. The modification is shown in Figure 3, which depicts three consecutive quanta, $q^{t-1}$, $q^t$, and $q^{t+1}$, each with a distinct $\beta$ value and quantum lengths $T_{act}^{t-1}$, $T_{act}^{t}$, and $T_{act}^{t+1}$. For the specified $PF_{loss}$, the expected execution time is denoted by $T_{exp}^{t-1}$, $T_{exp}^{t}$, and $T_{exp}^{t+1}$, respectively. Voltage/frequency scaling for $q^t$, $q^{t+1}$, and $q^{t+2}$ is performed at $t_1$, $t_2$, and $t_3$, respectively.

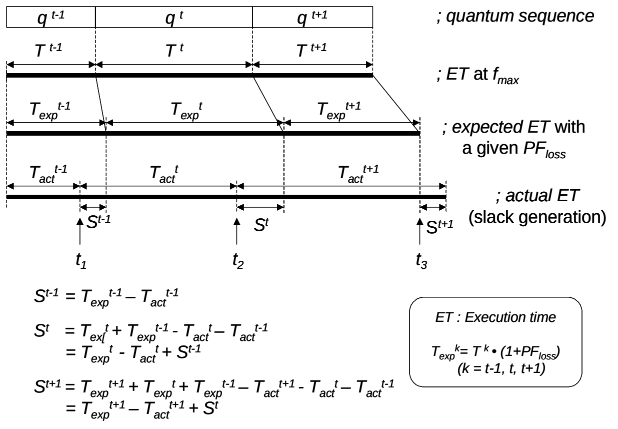

$q^{t-1}$
$q^{t}$
$q^{t+1}$
; quantum sequence
$T^{t-1}$
$T^{t}$
$T^{t+1}$
; ET at $f_{max}$
$T_{exp}^{t-1}$
$T_{exp}^{t}$
$T_{exp}^{t+1}$
; expected ET with a given $PF_{loss}$
$T_{act}^{t-1}$
$T_{act}^{t}$
$T_{act}^{t+1}$
; actual ET (slack generation)
$S^{t-1}$
$S^{t}$
$S^{t+1}$
$t_1$
$t_2$
$t_3$
$S^{t-1} = T_{exp}^{t-1} - T_{act}^{t-1}$
$S^{t} = T_{ex[}^{t} + T_{exp}^{t-1} - T_{act}^{t} - T_{act}^{t-1} = T_{exp}^{t} - T_{act}^{t} + S^{t-1}$
$S^{t+1} = T_{exp}^{t+1} + T_{exp}^{t} + T_{exp}^{t-1} - T_{act}^{t+1} - T_{act}^{t} - T_{act}^{t-1} = T_{exp}^{t+1} - T_{act}^{t+1} + S^{t}$
ET : Execution time
$T_{exp}^{k} = T^{k} \bullet (1+PF_{loss})$ $(k = t-1, t, t+1)$

**Figure 3: Compensating for the error due to misprediction of $\beta$**

When a frequency is chosen for the next quantum, there may exist some (positive or negative) slack time (i.e., the difference between $T_{exp}^{*}$ and $T_{act}^{*}$.) These slack times come about due to the misprediction of $\beta$ for the next quantum. With a positive (negative) slack, the frequency for the next quantum should be made smaller (larger) compared to the case of zero slack. For example, at time $t_2$, the actual execution time until $t_2$ is ($T_{act}^{t-1} + T_{act}^{t}$) which is less than the expected time ($T_{exp}^{t-1} + T_{exp}^{t}$), so there is a positive slack time $S^t = T_{exp}^{t} - T_{act}^{t} + S^{t-1}$. If $S^t$ is added in the calculation of the frequency for the next quantum $q^{t+1}$, then the error that occurred in the previous quanta can be compensated for. Eq. (9) for calculating the target frequency for next quantum is thus modified as follows:

$$f^{t+1} = \frac{f_{\max}}{1 + PF_{loss} \cdot \left[1 + \left(\beta^t + \frac{S^t}{PF_{loss} \cdot T_{act}^{t}}\right) \cdot \left(\frac{f_{\max}}{f^t}\right)\right]} \tag{10}$$

Notice that for positive (negative) slack $S^t$, the denominator will be larger (smaller) than the zero slack case, and hence the target frequency $f^{t+1}$ will be smaller (larger), which is of course the desired behavior.

## 5 Implementation

We implemented the proposed policy on a high-performance XScale-based testbed, which runs Linux (v2.4.17).

A programmable clock multiplier (PLL) in the XScale processor generates the internal CPU clock, which can be adjusted from 200 up to 733MHz in steps of about 66 MHz with the development-board speeds only available from 333 MHz and up. The lower bound results from a constraint to the memory bus speed, which is at 100 MHz in our system. The bus speed has to be less than a third of the CPU clock speed. This would yield a minimum speed of 333 MHz. Running the system at CPU speeds slower than 333MHz causes immediate halts. The main PCB of our testbed includes an on-board variable voltage generator, which provides suitable operating voltage at each clock frequency level. A D/A converter was used as a variable operating voltage generator to control the reference input voltage to a DC-DC converter that supplies operating voltage to the CPU. Inputs to the D/A converter were generated using a customized CPLD (Complex Programmable Logic Device). When the CPU clock speed is changed, a minimum operating voltage level should be applied at each frequency to avoid a system crash due to increased gate delays. In our implementation, these minimum voltages are measured and stored in a table so that these values are automatically sent to the variable voltage generator when the clock speed changes. Voltage levels mapped to each frequency are obtained through extensive measurements and summarized in Table 1.

For the measurements, the system has a 40K samples/second data acquisition system in which the voltage drop across a precision resistor inserted between the external power line and the "design under test" (DUT) power line is used to measure the power consumption as shown in Figure 4.

**Table 1. Frequency and voltage levels in the system**

| Frequency (MHz) | Voltage (V) |
|---|---|
| 333 | 0.91 |
| 400 | 0.99 |
| 466 | 1.05 |
| 533 | 1.12 |
| 600 | 1.19 |
| 666 | 1.26 |
| 733 | 1.49 |

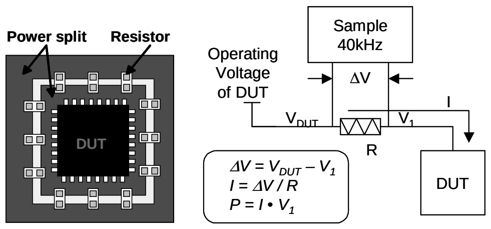


**Figure 4: Data acquisition system.**

As software works, we wrote a module in which the proposed policy is implemented and this module is hooked to the scheduler so that voltage scaling can occur during every context switch. Figure 5 shows the software architecture of DVFS implementation.

During the context switch, the PMU values for the previous process are read and the ideal frequency calculation for the next quantum is performed as described in section 4. A regression equation at each frequency is maintained for each process, which consists of no more than 5 long-type variables, resulting in little space overhead for implementing our DVFS policy. We measured the time overhead of our policy by using benchmark in the suite of the Lmbench [17] and found that the time overhead was about 100μsec. The original context switch time was also nearly 100 μsec. Although we almost doubled the context switch time, the overhead is still quite negligible in comparison to the quantum time of a few tens of millisecond. Our implementation supports a *proc-file* interface to the module such that the performance loss level and size of the window can be specified by writing the appropriate value to the this proc-file, which allows us to dynamically control the desired level of energy saving. Furthermore, the current values can be read from the proc-file interface. Another feature we have implemented to gain more accurate information (at the cost of higher overhead) is to measure the event values of PMU at every timer interrupt (1ms on our platform). This feature is disabled by default and is not exploited in the experimental results section.

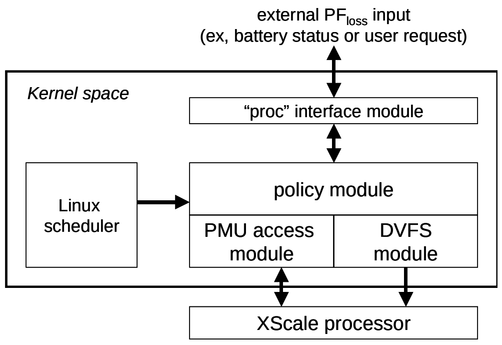


**Figure 5: Software architecture of our DVFS implementation**

## 6 Experimental Results

Our experiments are performed on the following applications including two common UNIX utility programs ("gzip" and "fgrep") and five representative benchmark programs available on the web [18]. They are summarized in Table 2. All the measurements are performed 10 times for each benchmark and the average performance loss and average energy saving values are reported. Size of the window, *N*, is set to 25 through exhaustive experiments. Based on the experimental results, it is found that *N* of 20 ~50 shows similar characteristics.

**Table 2. Summary of test applications**

| Benchmarks | Description |
|---|---|
| gzip | compressing a given input file |
| fgrep | searching for a given pattern in the files residing in a directory |
| math | floating-point calculations |
| bf (blowfish) | a symmetric block cipher with a variable length key from 32 to 448 bits |
| crc | 32-bit cyclic redundancy check on a file |
| djpeg | decoding a jpeg image file |
| qsort | sorting a large array of strings in ascending order |

Figure 6 represents the measured performance degradation with target performance loss ranging from 5% to 20% at steps of 5%. As seen in this figure, we obtained actual performance loss values very close to the target values for all programs (i.e., actual within 2% of the target) except for "fgrep" and "qsort" programs, which are memory-bound and $PF_{loss}$ of these are saturated to ~12%, corresponding to data in Figure 1. In Figure 7, actual power consumptions (including both CPU and DC-DC converter power) for two cases: (a) without DVFS and (b) with DVFS are reported when running "gzip". In case (a), the program is run at the maximum frequency (733MHz) and 10% target $PF_{loss}$ is given consistent with case (b). By applying the proposed policy, 52.1% of the CPU energy is saved at the cost of 11.6% performance loss. Measured energy savings for all benchmarks appear in Figure 8. From these measurements, we conclude that a CPU energy saving of more than 70% is achieved for memory-bound applications ("fgrep" and "qsort") with about 10% performance loss. The energy saving saturates after that, i.e., we cannot increase the amount of energy savings by tolerating a larger performance loss value. For CPU-bound applications, the degree of energy saving is smaller, but our approach allows a finely tuned energy-performance tradeoff. For example, in the case of "djpeg" program, we obtain a 42% CPU energy saving with a 20% performance loss constraint or a 26% energy saving with a 5% performance loss constraint.

## 7 Conclusion

In this paper, a regression-based DVFS policy for finely tunable energy-performance trade-off was proposed and implemented on an XScale-based platform. In the proposed DVFS approach, a program execution time is decomposed into two parts: on-chip computation and off-chip access latencies. The CPU voltage/frequency is scaled based on the ratio of the on-chip and off-chip latencies for each process under a given performance degradation factor. This ratio is given by a regression equation, which is dynamically updated based on runtime event monitoring data provided by an embedded performance-monitoring unit. Through actual current measurements in hardware, we demonstrated a CPU energy consumption of saving of more than 70% for memory-bound programs with about 12% performance degradation. For CPU-bound programs, 15~60% energy saving was achieved with fine-tuned performance degradation, ranging 5% to 20%.

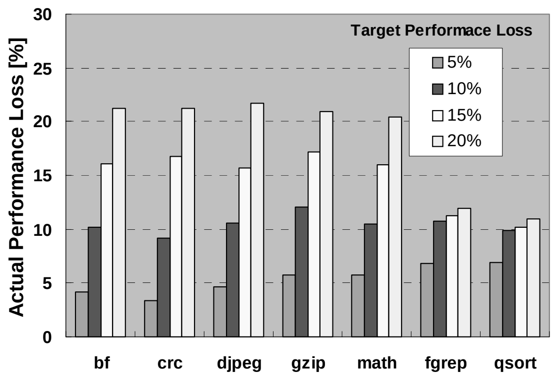
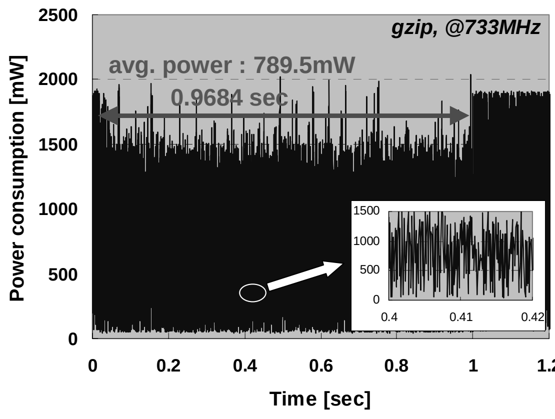
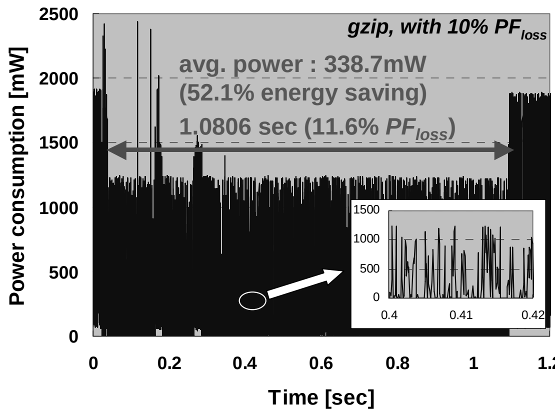


**Figure 6: Performance loss with different target values**



**(a) Without DVFS - at maximum frequency**



**(b) With DVFS - at a 10% performance loss constraint**

**Figure 7: CPU power consumption of with/without DVFS**

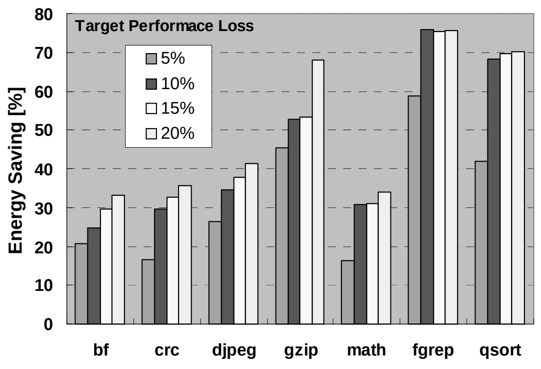


**Figure 8: CPU Energy saving for various application programs**

## References

[1] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," *IEEE Symp. on Low Power Electronics*, 1994, pp.8-11

[2] "Intel 80200 Processor Based on Intel XScale Microarchitecture," http://developer.intel.com/design/iio/manuals/273411.htm

[3] "Cruso SE Processor TM5800 Data Book v2.1," http://www.transmeta.com/everywhere/products/embedded/embedded_sefamily.html .

[4] F. Yao, A. Demers, and S. Shenker, " A Scheduling model for reduced CPU energy," *IEEE Annual Foundations of Computer Science*, 1995, pp.374-382

[5] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," *Proc. Int'l Symp. on Low Power Electronics and Design*, 1999, pp.197-202

[6] G. Quan and X. Hu, "Minimum energy fixed-priority scheduling for variable voltage processors," *Proc. Design Automation and Test in Europe*, March 2002, pp.782-787

[7] I. Hong, G. Qu, M. Potkonjak, and M.B. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processor," *Proc. of the IEEE Real-Time Systems Symp.* December 1998, pp.178-187

[8] D. Shin, J. Kim, and S. Lee, "Low-energy intra-task voltage scheduling using static timing analysis," *Proc. Design Automation Conf.*, 2001, pp. 438-443.

[9] S. Lee and T. Sakurai, "Run-time power control scheme using software feedback loop for low-power real-time applications," *Proc. Asia-Pacific Design Automation Conf.*, 2000, pp.381-386.

[10] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, "Profile-based dynamic voltage scheduling using program checkpoints in the COPPER framework," *Proc. Design Automation and Test in Europe Conference*, March 2002, pp.168-176

[11] C. Hsu and U. Kremer, "Compiler-directed dynamic voltage scaling for memory-bound applications," *Technical Report DCS-TR-498*, Department of Computer Science, Rutgers University, August 2002.

[12] C. Hsu and U. Kremer, "Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches," *Proc. Workshop on Power-Aware Computer Systems*, February 2002.

[13] D. Marculescu, "On the use of microarchitecture-driven dynamic voltage scaling," *Workshop on Complexity-Effective Design*, 2000.

[14] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC variation in workloads with externally specified rates to reduce power consumption," *Proc. Workshop on Complexity Effective Design*, 2000.

[15] A. Weissel and F. Bellosa, "Process Cruise Control," *Proc. Compilers, Architectures and Synthesis for Embedded Systems*, October 2002, pp.238-246

[16] J. Hennessy and D. Patterson, "Computer Architecture–A Quantitative Approach," 2nd, Morgan Kaufmann Publishers, 1996

[17] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Anaylis," *Proc. of the USENIX 1996 Technical Conf.*, January 1996, pp. 279-294

[18] http://www.eecs.umich.edu/mibench