A Framework for Continuously Adaptive DVFS

Vasileios Spiliopoulos
Uppsala University, Sweden
vasileios.spiliopoulos@it.uu.se

Stefanos Kaxiras
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

Georgios Keramidas
University of Patras, Greece
keramidas@ece.upatras.gr

***Abstract*—We present Continuously Adaptive Dynamic Voltage-Frequency Scaling in Linux systems running on Intel i7 and AMD Phenom II processors. By exploiting slack, inherent in memory-bound programs, our approach aims to improve power efficiency even when the processor does not sit idle. Our underlying methodology is based on a simple first-order processor performance model in which frequency scaling is expressed as a change (in cycles) of the main memory latency. Utilizing available performance monitoring hardware, we show that our model is powerful enough to i) predict with reasonable accuracy the effect of frequency scaling (in terms of performance loss), and ii) predict the energy consumed by the core under different V/f combinations. To validate our approach we perform high-accuracy, fine-grain, power measurements directly on the off-chip voltage regulators. We use our model to implement various DVFS policies as Linux "green" governors to continuously optimize for various power-efficiency metrics such as EDP (Energy-Delay Product) or ED^2P (Energy-Delay-Square Product), or achieve energy savings with a user-specified limit on performance loss. Our evaluation shows that, for SPEC2006 workloads, our governors achieve dynamically the same optimal EDP or ED^2P (within 2% on average) as an exhaustive search of all possible frequencies and supply voltages. Energy savings can reach up to 56% in memory-bound workloads with corresponding improvements of about 55% for EDP or ED^2P.**

I. INTRODUCTION

The power-aware architecture landscape has been dominated by techniques based on supply voltage and clock frequency scaling. Dynamic Voltage and Frequency Scaling (DVFS) offers great opportunities to dramatically reduce energy/power consumption by adjusting both voltage and frequency levels of a system according to the changing characteristics of its workloads. The great potential of DVFS in energy/power savings has been widely studied in a variety of research communities (from circuit to system designers) and has been extensively used in commercial systems as well. Intel XScale, AMD Mobile K6, and Intel Pentium M are typical low-power processors that feature DVFS management capabilities. Example processors from the high-performance area are the AMD Opteron quad-core and the Intel core i7 processor. At the operating system level, DVFS policies are coarse-grained, either based on user requests (the user requests a desired level or power/performance) or reacting to changes in the system load (when system load is low the processor is put in a low-power, low-frequency, low-supply-voltage mode).

While one can expect to lower power consumption by sacrificing performance, the promise of DVFS techniques lies in the exploitation of *slack* or "*idleness*." The objective is to take advantage of slack so that performance is affected little by frequency scaling while at the same time a cubic benefit in power consumption —with the help of voltage scaling— is achieved. Slack can appear at different levels and various approaches have been proposed for each level [10]. This work, is concerned with the instruction slack due to long-latency memory operations (off-chip memory accesses). In contrast to the OS-level DVFS policies available today that save power when the processor has little to do (system-level slack), we aim to save power even when the processor is busy executing programs (as long as these programs have *memory access slack* in them).

In our previous work [11], we developed two simple analytical models that are able to drive run-time DFVS decisions for aggressive superscalar OoO processors (e.g., Intel core i7 [4] or AMD Phenom II processors). The realization that inspired the development of these models is that scaling of the core frequency is nothing more than changing the memory latency measured in cycles.

Previous approaches [6,7,8,13,14,20,21,23] in the area rely on empirical models requiring large profiling, training and trial-and-error steps or significant compiler assistance [12]. In contrast, our models require minimal input and calculations [11] allowing for efficient run-time implementations. The reason for this is that our models are able to isolate the processor events that directly correlate DVFS to processor behavior.

The simple nature (minimal input and calculations) and the good accuracy of our models [11], inspired us to apply them in practice. While our previous work was conducted in a controlled simulation environment —a cycle accurate simulator augmented with power models— here, we discuss our experiences on real systems: the i7 Intel Nehalem [4], and the AMD Phenom II processors. Using run-time performance counters and accurate on-line power measurements for validation, we implemented OS kernel module governors that can dynamically predict performance and energy consumption of a workload at any target frequency and voltage with good accuracy. This allows various DVFS policies to be implemented and we show three such policies to optimize EDP or ED^2P. To our knowledge, these are the first practical, continuously adaptive DVFS Linux governors.

To validate the performance of the governor in terms of the accuracy of its decisions, we compare its resulting EDP or ED^2P to the optimal EDP or ED^2P that we can find with an exhaustive search of all the possible frequency and supply

voltage pairs for a given workload. Our results show that the run-time governor is within 2% of the best results of the exhaustive search.

Structure of the paper. Section II surveys related work. Section III provides an overview of our analytical DVFS models [11] and Section IV the implementation of Linux DVFS governors based on this model. Section V discusses power-related details of the i7 and Phenom II processors and our evaluation methodology, while Section VI presents the results of our evaluation of the adaptive DVFS governors. Section VII summarizes the paper.

II. Related Work

In this work our target is to provide a practical methodology for continuous and adaptive DVFS management in real processors. One of the first approaches in the area was by Grunwald et al [22]. The authors used a rather complex infrastructure (extra PCs for logging and process the required information and heavy modifications to Linux kernel) to control the power of the Itsy Pocket Computer (a research platform based on StrongARM). Their DVFS decisions were based on the fraction of time the Linux idle process enters the foreground for execution.

Almost all the related approaches for power estimation and management in real processors were motivated by the existence of a rich set of performance monitoring counters. The initial goal was to estimate the power consumed by the processor by selecting the appropriate monitoring events; subsequently DVFS policies were provided. However, all these approaches rely on empirically derived models requiring large profiling, preprocessing and/or training steps.

Joseph and Martonosi [16] use counter events to estimate the power of the Pentium Pro. Their approach requires 12 performance counters (but only two can be read simultaneously). They gather data for multiple benchmark runs forming an offline power model. Contreras and Martonosi [15] use five counters to estimate power for different frequencies on an XScale system. They also collect data from multiple runs in order to derive power weights for frequency-voltage pairs, and at a later step to create a parameterized linear model. A similar approach was developed by Rajamani et al. [18], but in this case the target was the Pentium M processor. None of these methods are intended for on-line use.

Lee and Brooks [17] predict power via statistical models. They build correlations based on hardware parameters identifying the most significant parameters to train their model. They perform an offline profiling of the design space and they estimate power based on random traces. A similar approach was followed by Goel et al. [20] which was initially based on the work of Singh et al. [19].

Isci et. al. [6,7] collect various performance counter events to create a history of program behavior and in particular "fingerprints" of a program's power behavior [6].

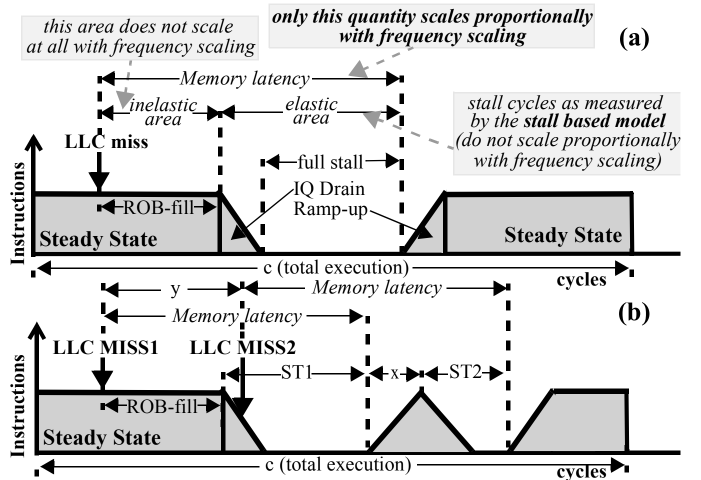

Figure 1. Useful instructions issued in the case of (a) an isolated and (b) overlapping LLC load misses.

More recently, Isci et al. extended their technique by predicting future power behavior [7]. In [8], Isci et al. extended their methodology to multicore processors. In this case also, the proposed model is prohibitively costly for run-time power estimation and optimization. It requires four complete program executions with different counter configurations to collect the necessary information. In [13,14], the authors follow a similar methodology with multithreaded applications and per-thread performance counters. The set of performance counters is empirically derived through extensive trial-and-error steps. Lastly, Shelepov and Fedorova [21] and Jimenez et al. [23] provide scheduling policies to reduce power in a heterogeneous multicore system by extracting architectural "signatures" of the applications. Again, such signatures must be known a priori through extensive offline analysis.

III. Interval-based Analytical Models for DVFS Management

In previous work [11], we show that a successful way to model DVFS in an OoO processor is to account only for the stall cycles introduced in the machine due to off-chip *non-overlapping* misses (Last-Level Cache or LLC misses). The idea is that only these misses directly correspond to the stall cycles that are affected by the processor's frequency. Based on this, we introduced a model, called *miss-based* model, which takes as input the number of stalls introduced in the machine due to non-overlapping LLC misses and outputs the execution time and energy under different frequencies with less than 1% (avg.) error. We also introduced a simpler model, called *stall-based* model, which is not able to distinguish overlapping of the LLC misses. The stall-based model still yields acceptable results (5% error on average). A deeper examination of this model shows that the extra error is introduced because the model disregards useful work

performed by the processor when a LLC miss occurs (i.e., from the occurrence of the miss to the point when no new instructions can enter the execution window or when all the instructions in the window are dependent on the miss).

Both our models derive from the interval-based performance model [2,9]. Intervals are marked by miss-events that upset the “steady state” execution of the program. A *miss-interval* starts with a miss-event (LLC misses in our case) and lasts until the IPC reaches again a steady state (a period related to the memory latency). Periods between miss-intervals are steady–state intervals. Figure 1 (a) shows the different areas of a LLC miss interval.

Stall-based Model. The simpler stall-based model takes as input the cycles which correspond to the *full stall+IQ Drain* areas and assumes that this quantity is equal to memory latency measured in cycles i.e., it disregards the ROB (Re-Order Buffer) fill area. Note that this area, measured in cycles, remains intact at all frequencies. The error of the stall-based model is apparent in the following assumption:

$$Stall_{cycles} = Mem_{lat} - ROB_{fill} \approx Mem_{lat}$$

The sum of stalls in overlapping misses (Figure 1.b) is also approximated to memory latency:

$$ST1 + ST2 = y + Mem_{lat} - ROB_{fill} - x \approx Mem_{lat}$$

In the stall-based model we assume that (i) the stalls generated by an isolated miss and (ii) the sum of stalls generated by overlapping misses are both approximately equal to the memory latency (in cycles). *Because memory latency is proportional to core frequency, these quantities are approximately proportional to frequency as well.* Consequently, the total number of stall cycles is approximately proportional to frequency, while the total number of non-stall cycles (steady state) is independent of frequency (measured in cycles).

Let c be the total cycles of a program execution and ST be the total number of stall cycles in max frequency f_{max}. In core frequency f_{max}/k the total number of cycles would be:

$$c_{new} = c - ST + \frac{ST}{k} \quad (3)$$

If the clock period under frequency f_{max} is T_{fmax}, then under frequency f_{max}/k the clock period is $k \times T_{fmax}$. The execution time under frequency f_{max}/k is:

$$t_{new} = c_{new} \times kT_{fmax} = (c \times k - ST \times k + ST) \times T_{fmax} \quad (4)$$

Using equation (4) we are able to predict the execution time under different frequencies by counting the total cycles and the stall cycles under frequency f_{max}. On the other hand, if the starting point is a frequency f where $f=f_{max}/l$, we can predict the corresponding values for the maximum frequency $c_{fmax}=c-ST+ST\times l$ and $ST_{fmax}=ST\times l$ and then substitute in Eq. 4 c_{fmax} and ST_{fmax} instead of c and ST.

The advantage of this model is that only in-core information is needed to predict performance under various frequencies. However, the model assumes that ROB-fill is negligible, which is a source of error especially in benchmarks characterized by little dependence between instructions and thus large ROB-fill times.

Miss-based Model. The more accurate miss-based model acknowledges that the whole miss interval equals memory latency and thus scales proportionally to frequency. Furthermore, it is able to recognize that *only the miss interval of the first miss in a cluster of misses scales with frequency, while the miss intervals of overlapping misses remain intact with frequency scaling.* Another way to express this is that if a miss occurs y cycles (Figure 1 b) after the initial miss it will also be serviced y cycles after the first miss is serviced *so the extra stall cycles introduced by the overlapping miss do not change with frequency.* When the miss that headed a cluster of overlapping misses is serviced, the next miss in line starts a new cluster *even if it overlaps with an outstanding miss from the previous cluster*.

The methodology followed in the miss-based model is similar to the stall-based model, but instead of stall cycles, the quantity that scales proportionally to the frequency is the number of clusters of misses multiplied by the memory latency. Unfortunately, there is no easy way in either the Intel i7 or the AMD Phenom II to account for the stalls introduced in the machine due to *non-overlapping* LLC misses. In other words, there are no specific performance counters to measure the *Memory-Level Parallelism* of the LLC misses, as pointed out also in [3]. We will not expand further on the miss-based model since it cannot be supported adequately in contemporary processors; instead, we refer the interested reader to [11] for further details.

IV. CONTINUOUS ADAPTIVE DVFS GOVERNOR

This section describes the high-level operation of our DVFS governor. The governor is called at regular intervals. Each time it is invoked it uses performance counter measurements collected during the previous interval to predict the performance/energy of the workload at all possible V/*f* pairs and select the most appropriate —based on the DVFS policy— V/*f* pair for the next interval. The assumption here is that measurements collected during an interval are valid inputs for our model to predict the next interval.

The actual V/*f* pair chosen for the next interval depends on the model prediction and on the DVFS policy we follow. In this work, we have implemented three DVFS policies which aim for *power-efficiency*:

- Optimize EDP (OptEDP): this policy from all the possible V/*f* pairs it selects the one that yields the minimum energy-delay product (EDP). EDP is a common power-efficiency metric that gives equal weight to both energy and performance (delay).
- Optimize ED^{2}P (OptED2P): this policy gives more emphasis on performance since it tries to minimize the product of energy and delay-square. Thus it will only sacrifice performance if the energy benefit is significant.

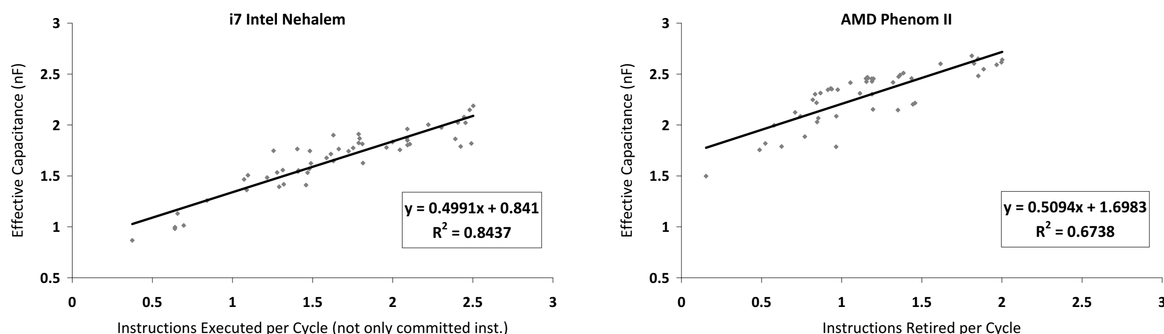

Figure 2. Relationship between effective capacitance and instructions executed (Intel)/retired (AMD) per cycle. The measurement was done at 1.6 GHz for Intel and at 2.1 GHz for AMD processors. The lower constant factor in the extracted equations indicates that the Intel processor is more clock-gated compared to AMD Phenom. The lower coefficient of determination (R^2) in AMD Phenom is due to the non-accounting of the speculatively executed instructions.

- Optimize EDP (ED^2P) under a performance penalty limitation, OptEDPlimit (OptED2Plimit): This policy minimizes EDP (ED^2P) but under the constrain that performance cannot be penalized by more than a user-selected factor (e.g., 10%).

Note that our methodology does not impose a particular DVFS policy but allows arbitrary policies to be constructed.

A. *Predicting Performance at Different f*

To predict the performance of a program at a different frequency than the one it is currently executing, the stall-based model discussed in Section III —in particular Eq. 4— requires as input the number of stalls strictly due to LLC misses. Given the available set of performance counters, in practice, we can only implement an approximation of the stall-based model. There are no performance counters for LLC stalls in either the Intel i7 or the AMD Phenom II. Instead, we approximate LLC stalls by taking all pipeline stalls (which may include stalls due to branch mispredictions, L1 cache misses, etc.) and the worst case for the LLC stalls assuming that no miss overlaps with any other. The worst case LLC stalls are simply the number of misses multiplied by the miss latency (in cycles).

The average error of our stall–based model implemented in the i7 is below 5% for all SPEC2006. We measure the error in predicting execution time for a very large frequency change: running each SPEC at the max frequency and predicting execution time in the minimum frequency or vice versa. In general, memory–bound programs exhibit larger prediction errors. This is an inherent property of the stall–based model, since this model ignores the ROB-fill effect. Similarly the error in performance prediction for the Phenom II is also below 5% for the SPEC2006. As we will see in the evaluation (Section VI), run-time governors achieve near-optimal decisions despite these approximations.

B. *Predicting Energy at Different f*

Predicting energy for the EDP and ED^2P calculations is more complicated since static energy is not negligible and needs to be taken into account in addition to dynamic energy. Our methodology in predicting the total energy of a program is the following: we measure (off-line) static power in idle state under all available different frequencies (see Section V for the details). We use a model to calculate dynamic energy based on performance counter information. On-line power measurements (see Section V) are used for validation of the predictions at the end of the next window.

Several models have been proposed that correlate various performance events to dynamic energy with good accuracy [8,13-18,20,21,23]. However, we are restricted by the small number of concurrently measurable events in the i7 and Phenom II. After measuring processor stalls (required for the performance prediction) we are left with only two performance counters in the i7 and the Phenom II. Related work [20] indicates that *instructions executed* in the i7 and *instructions retired* in the Phenom II, are the best, single-event correlations available to estimate dynamic energy.

We correlate instructions executed per cycle (i7) and retired per cycle (Phenom II) to the *effective capacitance* term (*activity factor* × *capacitance*) of the dynamic power equation. Frequency and supply voltage are known and unrelated to what is being executed. Figure 2 shows the derived relation for the two processors and the resulting correlation factor (R^2) using all SPEC 2006 as data points.

Figure 3 and 4 show the total energy prediction error (for the i7 and Phenom II respectively) compared to actual fine-grain power measurements (Section V). For these results we let the governor predict the total energy for a new interval based on the performance counter measurements of its preceding interval. At the end of the new interval the prediction is compared to real power measurements taken in its duration. In most cases (all SPEC2006), the error is below 5% for the i7 and 9% for the Phenom II. Exceptions are some exceedingly memory-bound workloads (e.g., mix1, mix3, and mix4, discussed in Section V) which yield very low IPC that was not accounted for in the correlation.

An alternative to using a performance counter model to make energy predictions, is to use actual power measurements if this capability is available. Our current

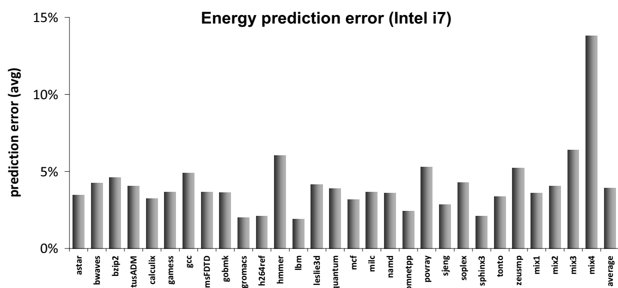

Figure 3. Run-time Validation for Intel i7 processor.

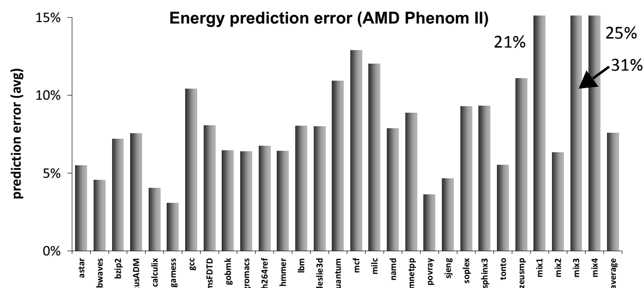

Figure 4. Run-time Validation for AMD Phenom II.

measurement setup easily allows this and upcoming processors (e.g., Intel Sandy Bridge [5]) will be able to measure power directly. We can measure total energy on-line, feed it back to the governor, subtract static energy (see Section V), and derive the relationship of dynamic energy to frequency. To predict dynamic energy at a new frequency, however, we need to know how the processor is clock gated. This information is not available [4]. We, therefore, consider two extremes: that of a fully clock-gated and that of a fully non-clock-gated processor. Dynamic energy in the former case is proportional to the square of the voltage ($E \sim V^2$), while in the latter case the energy should be computed according to the formula: $E \sim f \times V^2 \times t$. Our experimental findings reveal that the i7 core is not highly clock-gated, since the fully non-clock-gated hypothesis produces better results. Since we do not know the clock gating of these processors, even the measurement of actual power can only give us approximate predictions for the dynamic energy at different frequencies. Section V discusses our power measurement methodology that can be used for this purpose.

C. *Multicore DVFS*

Our discussion so far focused on DVFS policies for a single core running a single program. However, both the Intel i7 and the AMD Phenom II are multicore chips and can run simultaneously multiple programs. Furthermore, there is an important difference between them with respect to DVFS. The Intel i7 can scale V/*f* only for the whole chip (e.g., for all 4 cores) whereas the AMD Phenom II can scale *f* individually *per-core* but has to scale V for the whole chip —one of the reasons we examine both processors.

Concerning the DVFS policies in a multicore, the question that arises is how do we define EDP or ED^2P for multiple programs running simultaneously? In the case of the Intel i7 which can only select a single frequency for the whole chip we compute for each possible frequency the impact on performance (delay) for each running program and then use the average delay over all running programs to compute a chip-wide EDP or ED^2P. In the case of the AMD for each running program we compute the impact on its performance for each possible frequency and then use the average delay in the EDP and ED^2P calculation. For the OptEDPlimit and OptED2Plimit policies, the performance

```
# Offline: Compute static power for all frequency steps
#       and typical core temperature ranges
# Store results in table Pstatic[f,T] (indexed by freq.
#       step f and operating temperature range T)
green_governor() # invoked every q ms
# obtain the measurement of total energy
#       for the previous window:
  measured_P = obtain_measurement_from_USB
  measured_E = measured_P/t
# stalls[core]: perf. counter measurement of stalls
# IPC: perf counter measurement of instructions issued
#       per cycle (i7) or retired per c. (Phenom II)
  min_edp=inf
  for all frequencies f
    for all cores
      # invoke Eq.4:
      predicted_D[core]=stall-model(stalls[core],f)
    total_D = sum(predicted_D[core])
    predicted_E = Dynamic_model(IPC) + Pstatic[f,T]/t
# validate the prediction using real power measurements
# not part of the governor - used only for validation
#   error = difference(predicted_E, measured_E)
    edp = predicted_E * total_D
    if edp < min_edp
      min_edp = edp
```

Figure 5. Governor Pseudocode

limit applies to each of the applications (not to their total delay) —but other approaches are of course possible. Figure 5 shows the pseudocode for the governor that optimizes EDP (without any restrictions). The governor calculates performance and energy predictions for all possible V/*f* pairs and selects the one that yields the minimum EDP.

D. *Governor Overhead and Invocation Frequency*

The invocation frequency of the governors impacts two important factors: governor overhead and prediction accuracy. Overhead in our case consists of three components: running the code, reading performance counters, and potentially scaling V and *f*. The fewer invocations of the governor the less these overheads impact end performance.

At the time scale of the Linux scheduling quantum (10ms), these overheads are negligible. Accessing performance counters incurs a negligible cost [15,20,23] and DVFS transitions incur an overhead of tens of microseconds [24,25] —three orders of magnitude smaller than the scheduling quantum. Performance and energy prediction calculations are lightweight and thus not a concern.

Moreover, the time scale for the governor invocation affects prediction behavior. Governors invoked at very short

time intervals may react to short-lived changes in program behavior and erroneously change frequency and voltage for the next interval. At the other end, long intervals limit the optimization opportunities. In between, prediction accuracy is application dependent. Since we use a “last value” predictor (measurements taken during an interval are used for the next), the ideal interval length must track application program phases. But the time scale of program phases varies significantly and there is no single value that can accommodate all applications. The problem becomes more complicated if more sophisticated predictors are employed (e.g., using hysteresis in the prediction).

To balance these diverging requirements we explore a range of interval lengths that are multiples of the scheduling quantum (from 10ms to 1s). Below the scheduling quantum a separate entry into the Linux kernel is required and overheads start to become apparent. For the application mix we consider (SPEC2006) an interval 5× the Linux quantum (50ms) strikes the best balance between accuracy and overheads and we use this for the rest of the paper. A more formal methodology to select the governor interval is planned in our future work.

V. Experimental Setup

A. *Applications and OS*

We run our experiments on an Ubuntu Linux 9.10 system with the 2.6.31-22 kernel. The kernel is patched to enable our techniques to run as kernel modules. We use the entire SPEC2006 suite with *all* the ref. inputs.[1] We compiled the benchmarks with gcc 4.3 as 64-bits binaries and -O3 optimization. We use full benchmark runs to get a complete view of the benchmark behaviour (the benchmarks run for several minutes in our machines). Finally, the measurement of the performance counters runs as a kernel module, enabling counting in the OS. This way, no changes to the target applications are required and the overhead remains minimal (well below1%).

B. *Intel Core i7 and AMD Phenom II: main features*

The Intel core i7 is a quad-core CMP. Each core supports hyperthreading execution. The i7 Core family is enhanced with a special power-aware characteristic, called *Speedstep technology* [4] which allows fast run-time voltage/frequency scaling between 9 different steps, from 1.6 to 2.66GHz (i7 920). The i7 also supports various idle states, called C-states, in which it is possible to completely deactivate the clock and cut-off the power supply to a combination of cores (not available in the AMD). The quad-core AMD Phenom II supports 4 different frequencies, ranging from 800MHz to 3.2GHz. Each core can operate at an independent frequency but all cores are supplied by the same voltage (as needed by

[1] Due to space limitations, for the benchmarks with multiple inputs, we only include in the graphs the average over all inputs.

Number of Active Cores	2.66 GHz (Nominal Freq.)	1.6 GHz (Minimum Freq.)
4 cores	15.8W	7.6W
2 cores	10.4W	2.1W
1 core	2.6W	1W

Table 1: Power consumed by the i7 cores in the idle state.

Frequency (GHz)	Idle Power (W)
3.2	27.5W
2.5	17.11W
2.1	11.29W
0.8	5.7W

Table 2: Power consumed by the Phenom II in the idle state.

the core running at the highest frequency).

C. *Measuring Power and Energy*

Both i7 and Phenom II comprise of two main voltage islands: core (exec. and fetch units, OoO and paging logic, L1/L2 caches and branch prediction) and the uncore (L3 caches and memory controller). To isolate the core and the uncore power, we compute core power dissipation by directly measuring voltage and current from the off-chip voltage regulator (ADP4000 for i7 and ISL6323B for Phenom II) residing in the motherboard. While measuring voltage in both processors is straightforward, measuring current is highly dependent on the regulator design. In ADP4000 (i7’s regulator) there is a pin monitoring total output current. In Phenom II’s voltage regulator there is no current monitor pin, but output current can be estimated by measuring the voltage drop when the processor is under load. By hacking the motherboard (connecting wires to the appropriate pins) we are able to measure power while the processor was under normal operation. We use a sampling period of 10ms (our target is to provide OS-level optimizations so finer granularities will not provide more useful results). Power measurements are fed to the kernel OS using DLP-IO8, a USB analog–to–digital converter.

D. *Static Power*

When the processor is in the idle state, it consumes only static power since the clock is cut-off (the off-chip voltage regulator still provides voltage to the core). To get a full picture of how much power is consumed in the idle state, we deactivate different number of cores from the BIOS (available only in i7). The assessed idle power under different frequencies is gathered off-line by our kernel module. The stored power numbers are then used to predict the processor power at run-time. Tables 1 and 2 show idle power for both processors under different frequencies. Our methodology is able to take into account the dependence of static power to temperature (by sampling periodically the processor’s temperature sensors), but here we statically use the appropriate measurements for the operating temperature.

VI. Evaluation

Intel i7. In the first set of experiments we run a single

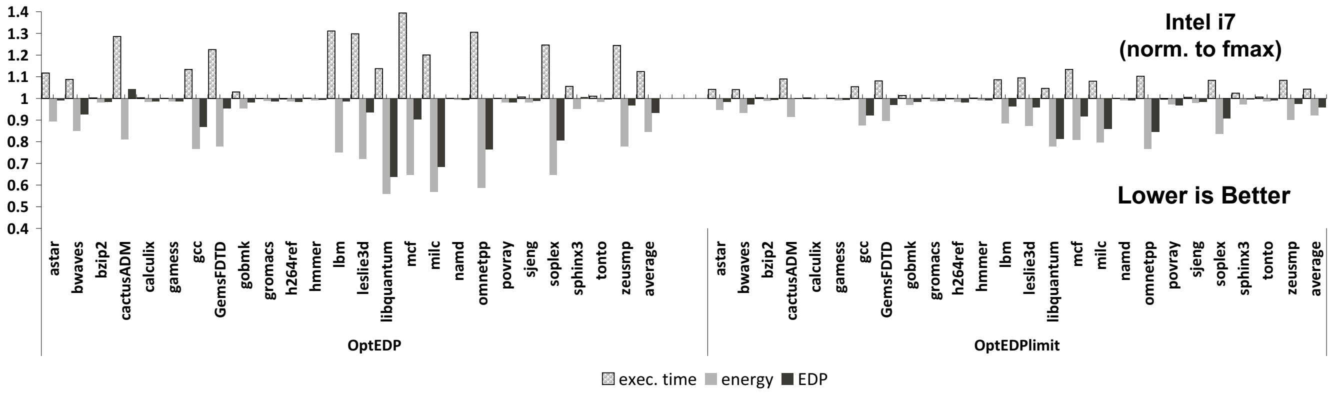

Figure 6. Performance, Energy and EDP (norm. to fmax) for the OptEDP and OptEDPlimit governors on the Intel i7.

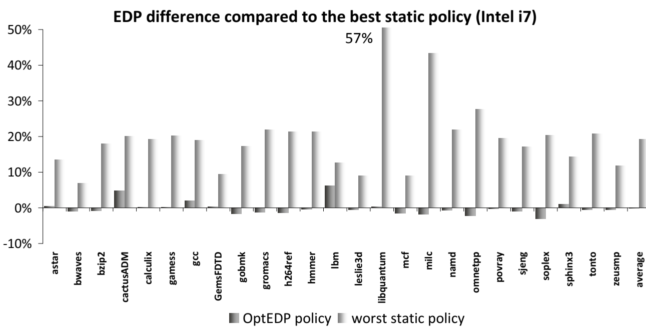

Figure 7. OptEDP governor comparison to best static and worst static profiling runs (Intel i7).

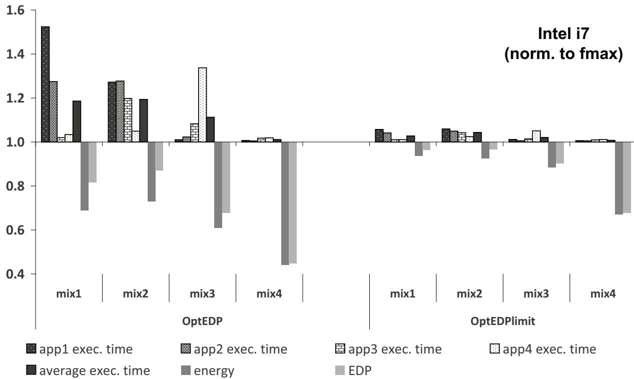

Figure 8. Multicore OptEDP and OptEDPlimit governor with 4 programs (Intel i7). Results are norm. to fmax.

SPEC2006 benchmark in one of the i7's cores and let the governor pick the V/*f* pair to optimize EDP without any constraints (OptEDP) and with a constraint of 10% on performance loss (OptEDPlimit). Figure 6 shows the performance impact, the energy savings (*measured*), and the resulting EDP for the two policies for all SPEC2006. In some CPU-bound programs there is a slight worsening in EDP, since the performance of such programs is very sensitive to frequency changes and any misjudgment in the governor's decisions impacts EDP negatively. This negative impact is reduced when we impose a restriction on performance loss. In contrast, in memory-bound programs (libquantum, mcf, mlic, etc.) EDP benefits can be substantial (exceeding 36% improvement in libquantum).

	APP1	APP2	APP3	APP4
MIX1	GAMESS	CALCULIX	LIBQUANTUM	MILC
MIX2	GAMESS	H264REF	SJENG	LIBQUANTUM
MIX3	LIBQUANTUM	MILC	OMNETMPP	CALCULIX
MIX4	LIBQUANTUM	LIBQUANTUM	MILC	MILC

Table 3: Mixed workloads. Shaded apps are memory-bound.

These results reflect the run-time decisions of the governor. But how far away are these results from the optimal EDP we could achieve with an exhaustive search of all possible V/*f* pairs? Figure 7 compares OptEDP (without constraints) to the "best static" case resulting from an exhaustive search. In the same figure we also show the "worst static" EDP, i.e., how bad one could do by selecting the worse possible V/*f* pair for each benchmark. For both OptEDP and "worst static" the percentage difference to the best static is shown. Note that in the static cases a single V/*f* pair is selected for the whole program while OptEDP dynamically changes V/*f* during the whole execution. In some cases this results in the governor beating the best static (negative error), since it can adapt to program phases.

Finally, Figure 8 shows the results of the OptEDP and the OptEDPlimit governor when running four mixed workloads in the quad-core i7 (each mix consists of four benchmarks shown in Table 3). Here, we compare our governor with the Linux OnDemand governor. Mix1 consists of two CPU-bound (gamess and calculix) and two memory-bound programs (milc and libquantum). While all four programs are running, the governor selects the lowest frequency and this is why gamess shows such a high performance penalty. gamess remains CPU-bound even with three other programs running at the same time. calculix also shows a significant performance penalty. The two memory-bound programs have small performance penalties —even though a low frequency is employed— but yield substantial energy savings. calculix is the longest-running program and still runs when all the other three programs have finished. At this point the governor realizes that it has a CPU-bound program and increases the frequency to the maximum. Overall, the governor yields close to 18% improvement in EDP over the

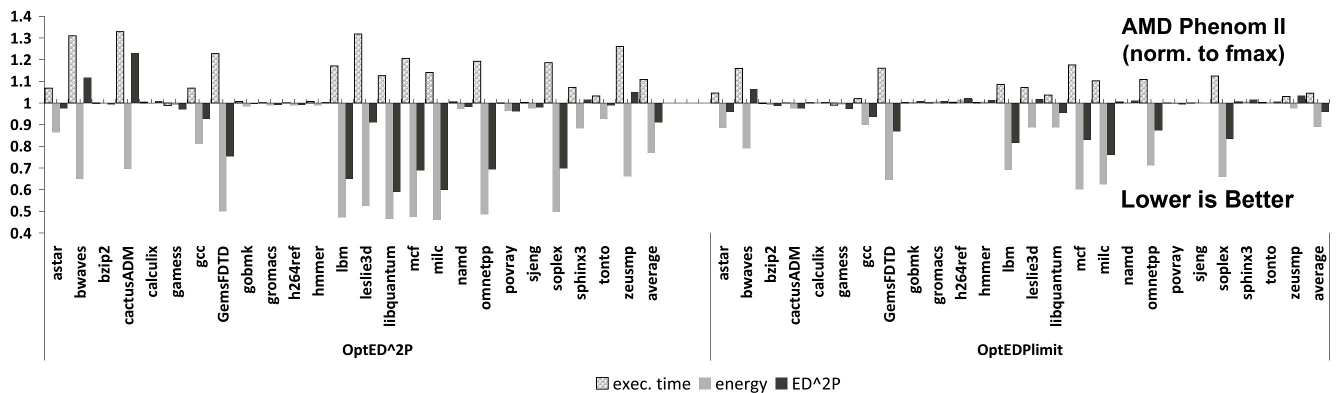

Figure 9. Performance, Energy and ED^{2}P (norm. to fmax) for the OptED2P and OptED2Plimit governors on AMD Phenom II.

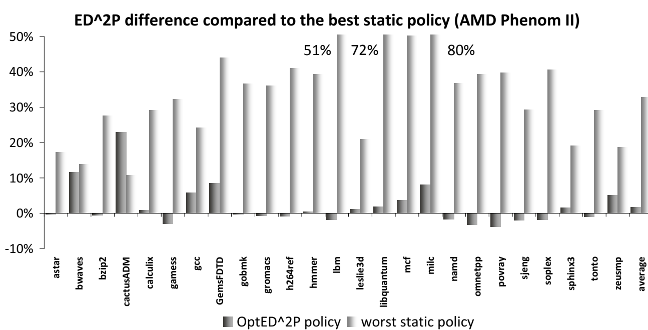

Figure 10. OptED2P governor comparison to best static and worst static profiling runs (AMD Phenom II).

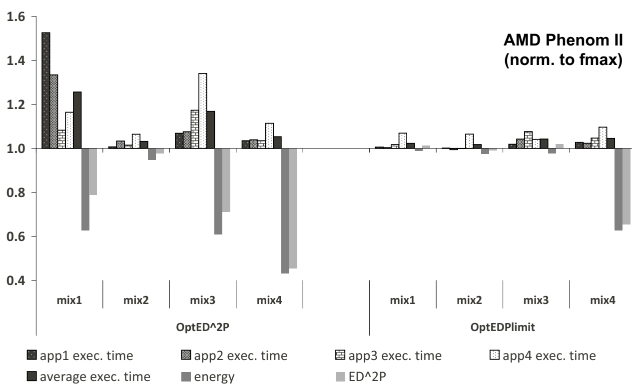

Figure 11. Multicore OptED2P and OptED2Plimit governor with 4 programs (AMD Phenom II). Results are norm. to fmax.

Linux OnDemand governor. Finally, mix4 is pure memory-bound workload (four memory-bound programs). In this case, OptEDP correctly selects to run all programs at the lowest available frequency with minimal performance overhead (below 2%), but with substantial energy benefits (56% reduction in EDP).

AMD Phenom II. For the AMD Phenom II we use the OptED2P governor which optimizes ED^{2}P. Optimizing EDP in the Phenom II is simply not very interesting: in the course of our study we discovered that all SPEC2006 programs have their optimal EDP at 2.1GHz! Even CPU-bound programs do not exhibit a better EDP at the highest frequency (3.2GHz). This is because of the high supply voltage (1.32V) required at high frequencies, the high static power dissipation, and the fact that the temperature of the Phenom II rises noticeably when running at full speed (which in turn increases static power). Using ED^{2}P, instead, gives us the opportunity to study the governor when emphasis is given to performance.

Figure 9 shows the resulting benefit in ED^{2}P compared to running with Linux OnDemand governor for single SPEC2006 benchmarks. Again we see substantial benefit for memory-bound programs (up to 41% for libquantum) but also some negative results for bwaves and cactusADM.

In bwaves, the governor errs because of the improper accounting of the LLC stall cycles (up to 24% performance prediction error) resulting in adverse frequency transitions. The situation in cactusADM is more complex. As it is depicted in Figure 10, even selecting the worst static V/f (for the whole program run) performs better than OptED2P governor. This is a non-intuitive behaviour because even if our OptED2P makes wrong decisions, the resulting ED^{2}P should be at least equal to the ED^{2}P of the worst case scenario (lowest frequency in this case). The reason is the following. Our OptED2P governor selects to alternate between the min and the max frequency at almost every interval. The frequent changes have an interesting side effect: the processor die is kept hot (as it runs at highest frequency) even for the small periods running at the lowest frequency. As a result, during the low frequency intervals, the static power dissipation is considerably high (due to the high temperature) resulting in higher chip power dissipation compared to the case when the lowest frequency is set for the whole run. Solving this issue requires our governors to take into account the thermal dissipation as an input which we plan as future work.

Considering the main advantage of the AMD Phenom II in terms of DVFS capabilities, the per-core frequency scaling, we expected it to be much more versatile than the Intel i7 when it comes to optimizing the ED^{2}P of multiple programs simultaneously. Our study, however, shows that

the limiting factor is the common supply voltage for the whole chip. Even though each core can be clocked independently, the supply voltage needs to be the highest voltage required by the highest frequency employed in the chip. As a result, the energy benefits between the Intel i7 (Figure 8) and the AMD Phenom II are almost the same in all mixes (except for mix2) invalidating the potential for higher energy savings due to the per-core DVFS.

Consider for example mix1 (Figure 11). What we would expect to happen with per-core DVFS is that the two CPU-bound programs (gamess and calculix) would run at the highest frequency while the two memory-bound programs (milc and libquantum) at the lowest. However, in the AMD Phenom II the two CPU-bound programs force the supply voltage to its highest level, raising the static power dissipation in the cores that run the memory-bound programs. Even though the two memory-bound programs run at low frequency they do not get the benefit in dynamic energy savings that comes from a reduced supply voltage. Our governor computes the OptED2P using the chip-wide supply voltage and arrives at a different frequency setting from the one sketched above: it chooses the lowest frequency for all programs even for the CPU-bound programs (as in i7). Thus, gamess experiences a significant performance penalty (exactly the same as in i7). The difference between Intel i7 and AMD Phenom II in mix2 comes from the fact that we use different power efficiency metrics for the two machines (EDP in i7, ED^2P in Phenom II). In this case, the OptED2P governor correctly picks to limit the performance penalty of the applications, reducing, however, the potential for higher energy benefits.

Overall, in the presence of memory-bound programs we see substantial reductions in total energy (with corresponding ED^2P improvements), which can exceed 50% for purely memory-bound workloads (mix4).

VII. Conclusions

In this work we describe our experience in creating a framework for accurate, continuously adaptive DVFS in real systems. We implement Linux DVFS governors based on our analytical DVFS models (in particular on the *stall-based* model [11]). Our aim is to optimize power-efficiency (expressed either as EDP or ED^2P) even when a processor is not lightly loaded and executes programs. To do this we rely on the inherent slack in memory-bound programs that can be exploited by DVFS. A unique characteristic of our approach (compared to previous approaches) is that it requires minimal input and calculations to make accurate performance and energy predictions for any target frequency. The input for performance predictions is approximated using available performance counters in contemporary processors. For energy predictions, static power is measured off-line and stored for run-time calculations; dynamic power is approximated using a model based on performance counters.

Based on these inputs, our green governors accurately predict the effect of voltage/frequency scaling with minimal calculations. The whole approach runs in kernel space thus introducing minimal timing overhead in the execution of the applications. The benefits can be substantial for memory-bound workloads, in some cases exceeding 55% improvement in EDP or ED^2P.

VIII. References

[1] D. Brooks et. al. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 2000.

[2] S. Eyerman et al. A mechanistic performance model for superscalar out-of-order processors. Transactions on Computer Systems, 2010.

[3] S. Eyerman, et al. A performance counter architecture for computing accurate CPI components. Int. Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[4] Intel Core™ i7-800 and i5-700 Desktop Processor Series, Intel, 2010.

[5] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, 2011.

[6] C. Isci et al. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. Annual Int. Symposium on Microarchitecture, 2006.

[7] C. Isci, et al. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. Int. Symposium on Microarchitecture, 2006.

[8] C. Isci and M.Martonosi. Run-time power monitoring in high-end processors: methodology and empirical data. Int. Symposium on Microarchitecture, 2003.

[9] T. Karkhanis and J.E. Smith. A first-order superscalar processor model. Annual Int. Symposium on Computer Architecture, 2004.

[10] S. Kaxiras and M. Martonosi. Computer architecture techniques for power-efficiency. Morgan & Claypool Publishers, 2008.

[11] G. Keramidas et al. Interval-based models for run-time DVFS orchestration in superscalar processors. Int. Conference on Computer Frontiers, 2010.

[12] G. Maglis, et al. Profile-based dynamic power voltage and frequency scaling for a multiple clock domain processor. ISCA, 2003.

[13] M. C. Maury, et al. Online power-performance adaptation of multithreaded programs using event-based prediction. Int. Conference on Supercomputing, 2006.

[14] M. C. Maury, et al. Prediction-based power-performance adaptation of multithreaded scientific codes. Transactions on Parallel and Distributed Systems, 2008.

[15] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. Int. Symposium on Low Power Electronics and Design, 2005.

[16] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. Int. Symposium on Low Power Electronics and Design, 2001.

[17] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. Int. Symposium on Architectural Support for Programming Languages and Operating Systems, 2006.

[18] K. Rajamani et al. Application-aware power management. Int. Symposium on Performance Analysis of Systems and Software, 2006.

[19] K. Singh, et al. Real time power estimation and thread scheduling via performance counters. Workshop on Design, Architecture and Simulation of Chip Multi-Processors, 2008.

[20] B. Goel, et al. Portable, scalable, per-core power estimation for intelligent resource management. Int. Green Computing Conf., 2010.

[21] D. Shelepov and A. Fedorova. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. Workshop on the Interaction between Operating Systems and Computer Architecture, 2008.

[22] D. Grunwald et al. Policies for dynamic clock scheduling. Int. Symposium on Operating System Design & Implementation, 2000.

[23] V. Jimenez et al. Power and thermal characterization of POWER6 system. Int. Conference on Parallel Architectures and Compilation Techniques, 2010.

[24] W. Wu et al. Multi-Phase buck Converter Design with Two-Phase Coupled Inductors. Int. Applied Power Electronics Conference and Exposition, 2006.

[25] T. Burdand and R. Brodersen. Design issues for dynamic voltage scaling. Int. Symposium on Low Power Electronics and Design, 2000.