Power-Performance Adaptation in Intel Core i7

Vasileios Spiliopoulos[1], Georgios Keramidas[2], Stefanos Kaxiras[1], and Konstantinos Efstathiou[3]

[1]Uppsala University, Sweden
[2]Industrial Systems Institute, Greece
[3]University of Patras, Greece

Abstract. In this paper, we describe our experiences in building a framework for power/performance run-time management for the Intel core family. Our underlying methodology (in contrast to previous work which relied on empirical models) is based on a simple processor performance model in which frequency scaling is expressed as a change (in cycles) of the main memory latency. Based on this model and utilizing performance monitoring hardware, the proposed model is shown to be powerful enough to i) describe and explain how Intel processors are affected by frequency scaling with respect to workload behavior, ii) predict with reasonable accuracy the effect of frequency scaling (in terms of performance loss), and iii) predict the energy consumed by the core under different *V/f* combinations by directly measuring from the off-chip voltage regulator the power consumed by the core. Our long-term plans include integrating in the proposed framework various power-aware OS/application-driven DVFS policies. As a first step towards this direction, we show our experimental methodology to justify the power/performance measurements and verify the correctness of our framework in which any target DVFS policy can be embedded as kernel module.

1 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.

In general, the heart of DVFS techniques is the exploitation of the system *slack* or "*idleness*." Their 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 [8]. Slack can appear at different levels and various approaches have been proposed for each level. According to [8], DVFS decisions can be taken at: i) the system level based on system slack, ii) the program level based on instruction slack, and iii) the hardware level based on hardware slack. More details about the criteria used to devise this categorization can be found in [8]. In this work, we are concerned with the instruction slack due to the long latency memory operations (off-chip memory accesses).

In our previous work [9], we developed two simple analytical models that are able to drive run-time DFVS decisions for aggressive superscalar OoO processors. These models work at the microarchitectural level and their target is to exploit the slack due to the long-latency, off-chip memory operations. *The realization that inspired the development of these models was that core frequency is nothing more than changing the memory latency in cycles.* This conceptual view of frequency scaling significantly simplifies the DVFS management decisions even for highly-aggressive, highly-pipelined, dynamic processors (e.g. the Intel core i7 [3]).

Previous approaches [4][5][6][11][12] in the area rely on empirical models requiring large profiling, training and trial-and-error steps or significant compiler assistance [10]. For example, the model proposed in [6] is prohibitively costly for run-time power estimation and optimization. It requires four complete program executions with different counter configurations, in order to collect the necessary information. In contrast, our models require minimal input and calculations [9]. The reason for this is that our models are able to acknowledge and isolate the processor events that directly correlate to DVFS processor behavior. Consider for example, the penalty of a branch misprediction. This penalty (measured in cycles) will remain intact no matter what the frequency is, because a branch misprediction involves only in-core operations. The penalty of in-core operations is always the same (measured in cycles) during frequency scaling [9].

The simple nature (minimal input and calculations) and the high accuracy of our models [9], inspired us to move one step forward. While our previous work was conducted in a cycle accurate simulator (equipped with the appropriate power models), in this work we provide our experiences and application results in applying those models in a real-life processor: the i7 Intel Nehalem core [3]. Testing research ideas in real processors was motivated by the integration of a rich set of performance monitoring counters which resides in almost all modern processors. It is well known that cycle-accurate simulations are very time consuming and their accuracy is a subject of considerable debate. Consider for example thermal studies where it takes a long time for processors to reach equilibrium thermal operation points. Live measurements allow a complete view of operating system and I/O effects and many other aspects of “real-world” behavior, often omitted in simulation. However, measuring live, running complex systems (i7 920 is a quad core SMT CMP) and relating measured results to overall system hardware and software behavior is not so straightforward as in a simulator, because many details are omitted from the computer vendors. *As a result a systematic approach is required to reverse-engineer the hardware details of the target processors.*

In this work, we provide a framework for power and performance run-time management for the Intel processors. Our framework can be formed as a basis for future power-aware research. As a first step towards this direction, we provide our experimental methodology to justify the power/performance measurement and verify the correctness of our framework in which any target DVFS policy can be embedded in the OS as kernel module. Some of the points, we try to shed light on, are: i) How much power (static and dynamic) is consumed by the core and the uncore areas of the processors? ii) How Intel processors are affected by frequency scaling with respect to the behavior of the applications? iii) Is the performance monitoring hardware appropriate for power-oriented optimizations? iv) How much clock-gated is the i7 core?

To quantify the robustness of the proposed framework, in this work we investigate the following scenario: we run an application in a specific V/f point collecting performance and power measurements (see Section 4 and 5). Based on those measures, we evaluate our methodology in predicting the performance and energy characteristics of the application in any given frequency/voltage combination (specified for example by the user or the OS).

Structure of the paper—Section 2 provides an overview of our previous work [9]. Section 3 discusses power-related details of the i7 core and presents our measurement methodology. Section 4 analyses the effect of frequency scaling in i7 core with respect to power "behavior" of the applications. Section 5 provides our experiences in predicting the energy under different *V/f* points. Section 6 outlines the presented methodology and discusses our future work. Section 7 concludes the paper.

2 Interval-based Analytical Models for DVFS Management

In our previous work [9], we showed that a successful way to model DVFS management in an OoO, dynamic 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 LLC misses and outputs the execution time —and the energy— under different frequencies with less than 1% (avg.) error. We also introduced a more simplified model, called stall-based model, which is not able to distinguish pipelining of the LLC misses (i.e., it accounts for stalls for both isolated and overlapping misses). The stall-based model still yields acceptable results (5% on avg.). 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 until no more instructions can enter the execution window or all the instructions in the execution window are dependent on the pending miss). However, the latter model offers a great potential: it can be used in real-life processors (e.g. the i7 core), in contrast to the miss-based model (the current hardware monitoring events are not able to distinguish between overlapping and isolated misses [2]).

Our modelling methodology is inspired by the interval-based performance model [1][7]. 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. *The realization that drives this work is that core frequency scaling in these models is nothing more than changing the memory latency in cycles.* Figure 1.a shows the different areas of a LLC miss interval. The 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 — it disregards the ROB fill area. Note that this area, measured in cycles, remains intact in all frequencies. The error of the stall-based model can be seen in the following formula:

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

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

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

The conclusion is that *the sum of stall cycles is proportional to memory latency and thus proportional to frequency scaling.* On the other hand, non-stall cycles remain

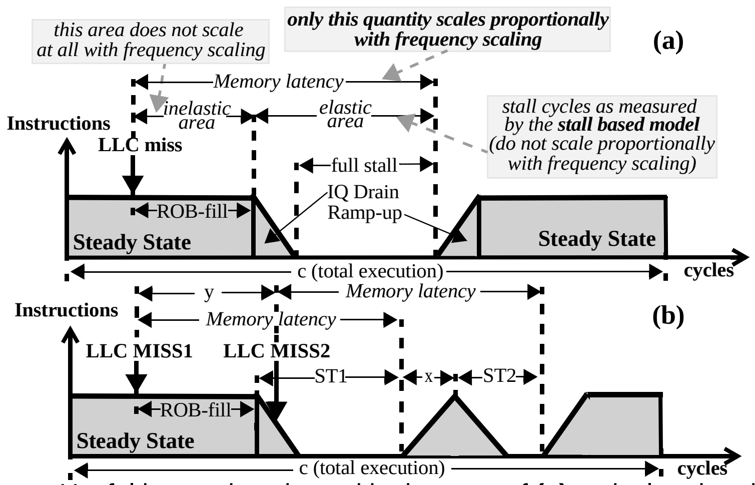

Figure 1 Useful instructions issued in the case of **(a)** an isolated and **(b)** overlapping LLC load misses.

intact. With these observations in mind, extracting the formula for predicting execution time is straightforward (more details can be found in [9]).

The miss-based model acknowledges that the 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 by 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. More information can be found in [9].

3 How and What to Measure in Intel Core i7

3.1 Intel Core i7: main features

Intel core i7 is a quad-core CMP. Each core supports hyperthreading execution. i7 Core family is enhanced with a special power-aware characteristic, called *Speedstep technology* [3] which allows run-time voltage and frequency scaling between 9 different steps, from 1.6 to 2.66GHz (i7 920). i7 supports also various idle states, called C-states, in which it is possible to completely deactivate the clock and cut-off the power supply for a combination of cores to reduce static and dynamic power consumption.

3.2 Performance Counters in i7

i7 core offers a wide range of hardware monitoring event counters. Table 1 shows the performance counters used in this work. One of our main problems in this work was

Performance Counter	Description
UOPS_EXECUTED.CORE_STALL_CYCLES	cycles no insts are executed in the processor
L2_RQSTS.LD_MISS	load requests that missed L2 cache
LLC_MISSES	last level cache misses
BRANCH_MISSES.RETIRED	mispredicted branches
UOPS_EXECUTED.PORT015	micro-ops executed in ports 0, 1 or 5
UOPS_EXECUTED.PORT234	micro-ops executed in ports 2, 3 or 4
UNHALTED_CORE_CYCLES	cycles core not halted

Table 1: The Intel i7 core hardware events selected for this work.

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 2: Power consumed by the i7 cores in the idle state.

that there is no performance counter in i7 core to account for the stalls introduced in the machine due to LLC non-overlapping misses (Section 2). In other words, there are no specific performance counters to measure the *Memory-Level Parallelism* of the LLC misses (also pointed out in [2]). As a result, we could not use our highly accurate miss-based model. Therefore, we used an approximation of the stall-based model in order to predict the performance under different *f* points using the counter event mentioned in line 1 in Table 1. Finally, all the other performance counters listed in Table 1 are used only for cross-checking our results (not to predict the effect of frequency scaling) and to gain a better understanding of the behavior of the running applications.

3.3 Measuring Power/Energy in i7

i7 core comprises 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, memory controller and QPI). In order 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) residing in the motherboard by identifying two pins of interest: the pin that supplies the voltage to the core and the pin monitoring the total output current of the regulator. By hacking the motherboard (connecting wires to these pins) we were able to measure power using an oscilloscope 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 useful results). The power measurements can be easily fed to the kernel OS using DLP-IO8, a USB analog–to–digital converter. Our future work includes utilizing this information (in the kernel level) to drive application/OS-driven DVFS policies.

3.4 Static Power in i7

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. The assessed idle power under different frequencies is gathered by our kernel module. The collected power numbers can be then used for predicting the processor power at run-time. Table 2 shows idle power for different number of cores under maximum and minimum frequency.

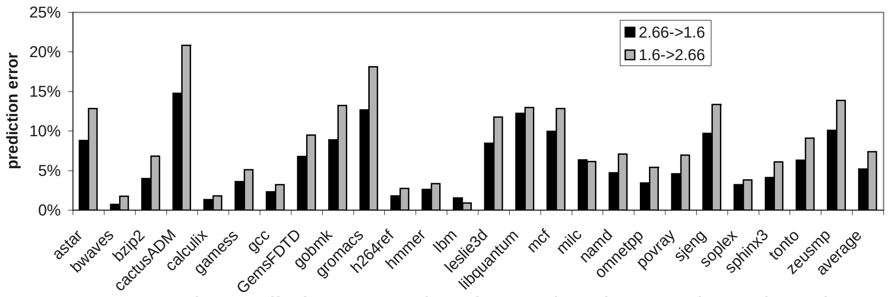

Figure 2 Error in predicting execution time using the stall-based model.

3.5 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. 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 behavior (the benchmarks run for several minutes in our machine). 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 timing overhead during execution remains minimal (less than 1%).

4 Predicting Performance for Different *f* Points

Figure 2 shows the absolute error of predicting the execution time using our stall–based model in i7 core for a large frequency step: running the program in the nominal frequency and predicting the execution time in the minimum frequency (black bars) — the grey bars represent the reverse scenario. Due to space restrictions, for the benchmarks with multiple inputs, we present in Figure 2 the average error over all inputs. To further analyze the results, we classify the benchmarks into three categories: *CPU–bound*, *memory–bound*, and *intermediate* or *mixed* category. This categorization is performed as follows: when the frequency is scaled from 2.66 to 1.6 GHz, a purely CPU-bound program will suffer an increase in its execution time of 66.67%, but due to memory accesses this penalty will be smaller. Based on this, a program with performance penalty of more than 55% (when scaling the frequency from max to min) is CPU-bound, a program with penalty less than 35% is memory bound while the rest of the benchmarks fall into the intermediate category.

In general, the more memory–bound a program is, the more the increase in the prediction error. This is an inherent property of the stall–based model, since this model ignores the ROB-fill effect. In the rest of this section we explain errors on a per-benchmark basis. To be able to delve into details about the power "behavior" of each benchmark, we also gather information for all the events listed in Table 1.

CPU bound— This category contains the following benchmarks/inputs: *astar_2*, *bwaves*, *bzip2* (6 inputs), *cactusADM*, *calculix*, *gamess* (3 inputs), *gobmk* (5 inputs), *gromacs*, *h264ref* (3 inputs), *hmmer* (2 inputs), *namd, povray*, *sjeng*, *sphinx3, tonto and zeusmp*. The errors in predicting the execution time in this category for 17 benchmarks are below 5%. Clear exceptions are *cactusADM* (14.8% error), *gromacs* (12.7%), and *astar_2* (10.9%). To understand and explain these errors, Table 3 shows

CATEGORY	BENCHMARK	TIME PENALTY (%)[A]	STALLS	L2 MISSES	L3 MISSES	BRANCH MISPRED	IPC[B]	**ERROR (%) FROM 2.66 TO 1.6 GHZ**
CPU BOUND	CACTUSADM	63.4	356	2.7	2	0.02	2	**14.8**
	GROMACS	65.8	287	2.2	0.01	4.96	1.97	**12.7**
	ASTAR_2	67.1	234	6.36	0.09	23.66	2.28	**10.9**
	ZEUSMP	57.4	348	3.22	2.5	17.84	1.83	**10.1**
	SJENG	63.3	260	0.96	0.5	30.7	2.26	**9.7**
	BZIP2_3	60.8	127	11.4	0.02	28.4	2.33	**2**
	H264REF_1	66.1	37	1.35	0.05	13.8	2.49	**1.4**
	GAMESS_2	65.3	46	0.16	0	27.2	2.5	**1.3**
	CALCULIX	65.8	39	1.12	0.1	41	2.54	**1.3**
	BWAVES	58.4	134	1.04	1	12.47	2.39	**0.7**
MIXED	MCF	44.8	518	25	10.4	9.1	1.47	**10**
	LESLIE3D	52.9	378	4.1	3.6	0.36	1.9	**8.5**
	GEMSFDTD	52.4	353	7.6	4.3	0.3	1.67	**6.8**
	ASTAR_1	52.1	357	10.8	3.9	14.4	2.13	**6.7**
	GCC_2	54.6	304	6.72	1.67	17.1	2.13	**6**
	LBM	48	231	1.87	2.88	6.75	1.78	**-1.5**
	GCC_5	37.4	394	4.35	5.36	7.15	2.11	**-1.3**
	GCC_3	35.9	421	3.34	4.45	7.26	2.15	**-1.2**
	GCC_4	43.3	336	4.52	5	10.15	2.14	**-0.2**
	GCC_6	38.4	405	4.21	5.25	6.94	2.14	**-0.1**
MEMORY BOUND	LIBQUANTUM	17.4	483	3.27	5.58	4.97	1.6	**-12.2**
	MILC	22.5	519	10.9	12.9	3.19	1.99	**-6.4**
	GCC_8	28.5	472	3.95	4.89	5.96	2.18	**-4.1**
	OMNETPP	33	565	14.7	7.35	6.49	2.04	**3.4**
	GCC_7	32.2	480	3.18	5.24	7.11	2.1	**-1.3**

Table 3: Performance counter events (per 1K cycles) and prediction error for three benchmark categories: CPU bound, mixed, and memory bound category.

a. Performance penalty when the frequency is scaled from 2.66 GHz to 1.6 GHz.
b. Instructions per non-stall cycles (indicates Instruction Level Parallelism).

the results for the full list of the performance counters for representative cases. By cross comparing the results, we are able to explain the behavior of each benchmark.

The error in *cactusADM* is due to the increased number of stalls generated by L2 and LLC misses and the low IPC. Although only 2 LLC misses per 1k cycles occur, the penalty for a *fmax* to *fmin* transition is 63.4% which means that these L3 misses overlap either with each other or with L2 misses, thus few of the stalls counted are *strictly* due to LLC misses. *gromacs* shows a similar behavior. The increased number of stalls are due to L2 misses, branch mispredictions and low IPC. On the other hand, *astar_2* has a quite high IPC which "hides" the stalls due to the large number of L2 misses thus the prediction error is smaller (10.9%). *zeusmp* lies on the border between CPU-bound and mixed categories, so the 2.5 LLC misses per 1k cycles introduce some stall cycles but L2 misses, branch mispredictions and low IPC also introduce stall cycles resulting in an error of 10.1%. Finally, *sjeng* has few misses but many branch mispredictions and as a result a 9.7% error. In general, benchmarks with few miss events and high IPC exhibit smaller errors. For example, *h264ref1* and *calculix* have about 1 L2 miss per 1k cycles and 13.8 and 41 branch mispredictions respectively, but due to high ILP (indicated by high IPC) few stalls are introduced in the machine resulting in low errors. Similar behavior is reported by *gamess_2*. *bwaves* approaches the mixed category, but the extra stalls introduced by L2 misses and branch mispredictions boost the model to compensate for its inherent inability to account for the ROB-fill area (this phenomenon will be further explained in the next categories).

Mixed category—This category includes: *astar_1*, *gcc* (6 inputs), *gemsFDTD*, *lbm*, *leslie3d*, *mcf* and *soplex* (2 inputs). As we move towards more memory-bound programs, we observe that the prediction error becomes smaller. This is because now the stall cycles introduced by L3 misses become a larger part of the total stall cycles and what we measure is closer to what we should measure according to the stall-based model. *mcf* yields the largest prediction error in this category (10%) due to the large amount of L2 misses (25), the branch mispredictions (9.1) and the strong dependencies between instructions illustrated by the low IPC (1.47). Although there are many L3 misses (10.4), *mcf*'s penalty is only 44.8%, which means that most of them are not performance critical (overlapping misses). *leslie3d* and *gemsFDTD* are more CPU-bound compared to *mcf* and the prediction error is smaller due to the reduced amount of miss events (4.1 and 7.6 L2 misses and 0.36 and 0.3 branch mispredictions respectively). *astar_1* has about the same penalty with *leslie3d* and *gemsFDTD* and more miss events (10.8 L2 misses and 14.4 branch mispredictions), but it also has larger IPC (2.13) which means that the processor is able to keep executing instructions when a miss event occurs and thus the majority of stalls measured are due to L3 misses. Similar to *astar_1* is *gcc_2* which has a slightly smaller prediction error due to the reduced amount of L2 misses.

Until now we explained how miss events other than L3 misses pollute our measurements. As the program becomes more memory-bound, L3 misses become the governing factor in stalls and another source of inaccuracy arises: neglecting the ROB-fill area. Although this is an inherent problem of the stall-based model, the way we measure stalls improves accuracy because the extra stalls measured due to other miss events compensate for the non-measured ROB-fill area. The negative error indicates that the stall cycles are underestimated. So in *lbm*, L2 misses, branch mispredictions, and low IPC produce extra stalls which reduce the error. The error would be even smaller if more stalls were measured. This is the case for *gcc_3*, *gcc_4*, *gcc_5*, *gcc_6*. The stalls introduced by L2 misses and branch mispredictions result in small prediction error, less than 1.5%.

Memory bound category—This category includes 5 benchmarks. The largest prediction error is observed in *libquantum* (12.2%). The low increase in execution time (17.4) between the maximum and the minimum frequency indicates that *libquantum* is heavily memory bound, although L3 misses are not that many. This means that all or most of them are performance critical (isolated). *libquantum* has a few other miss events which reduce the error (negative error means that stalls are underestimated and extra stalls reduce prediction error). *milc* is slightly more CPU-bound compared to *libquantum* but also has more L2 misses, so the extra stalls reduce prediction error to 6.4%. *gcc_8* is more CPU-bound and the prediction error is improved. *omnetpp* and *gcc_7* have about the same penalty, but differ in the sign of prediction error. *omnetpp* has many L2 misses which results in overestimating the stalls and thus positive errors. On the other hand, *gcc_7* has fewer L2 misses and thus the prediction error is negative.

5 Predicting Energy for Different *f* Points

Our methodology in predicting the dynamic energy of a program is the following: we measure static power (power in idle state) under all available different frequencies. To

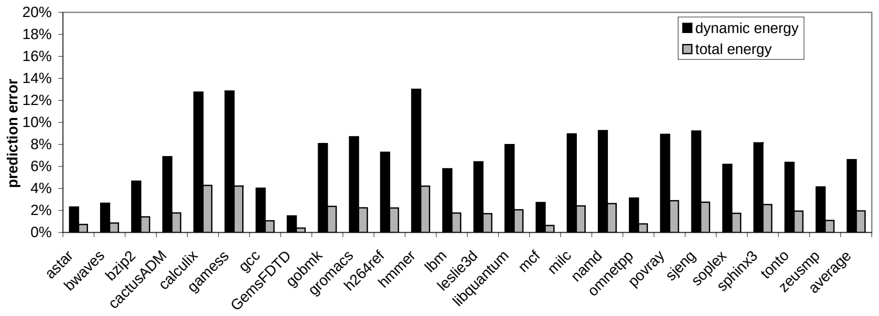

Figure 3 Error in predicting dynamic and total energy in i7 core.

calculate the run-time dynamic energy of a program, we subtract from the total energy the product of execution time and static power for the corresponding frequency. Finally, to predict the dynamic energy consumed in a new frequency, we tested the two extremes of a fully clock-gated and 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: $E \sim f*V^2*t$.

Our experimental findings reveal that the i7 core is not highly clock-gated, since the fully non-clock-gating scenario produced better results. Especially, in memory bound programs, in which the clock gating is expected to save more energy due to the long stall intervals, the results for the fully clock-gated case were even worse compared to the non clock gated scenario. A more accurate model could be derived if we knew exactly the processor clock gating map. However this information is not available [3]. Total energy can be predicted by adding to the predicted dynamic energy the product of the new static power and the execution time. Figure 3 shows our results. The grey bar shows the results for total (dynamic and static) energy prediction, while for clarity reasons we also depict the results for dynamic energy prediction (black bar). Figure 3 depicts the errors when all cores are active. Maximum and average error are 14% and 8% respectively for dynamic energy prediction, while the errors in total energy prediction are less than 5% and 2% respectively. In case that fewer cores were activated from BIOS, the dynamic energy prediction error would be the same but the total energy prediction error would be between the current value and the dynamic energy error for each program.

6 Overview of the Methodology and Future Work

The study presented in the paper shows a development stage of our work in building a strong framework for power/performance runtime management for the Intel processors. Our view is that this framework can form the basis for future power-related research. A unique characteristic of our approach (compared to previous approaches) is that it requires minimal input and calculations. The required inputs are: a single performance counter (line 1 in Table 1) and the power consumed by the processor collected by monitoring the off-chip voltage regulator. Both inputs are gathered at run-time by our kernel module. The idle power consumed by the processor in all frequencies (9 values in i7) is also stored in our kernel module. Based on those inputs, our kernel module is able to predict the effect of frequency scaling with minimal calculations (presented in our previous work [9]). The whole approach runs in kernel space thus introducing minimal timing overhead (less than 1%) in the execution of the applications.

In this work, we focus on the following scenario: we perform a single whole run of an application and using our methodology we predict the execution time and energy consumed by the application in different frequencies (i.e. the core frequency is kept constant during the whole execution). We are currently extending this work towards a window-based approach in which our kernel module applies different V/f points at runtime aiming to optimize different energy-efficient metrics (e.g EDP, application/OS energy metrics) in analogy to our runtime DVFS management in a simulated environment [9].

7 Conclusions

We described a hardware-specific implementation of the stall-based model proposed in our previous work [9]. In order to explain how the model performs in the i7 core, we attempted a qualitative analysis of how prediction accuracy is affected by various benchmarks' behavior. Our experimental results show that the execution time of the applications can be predicted for various frequency scaling steps —even for the extreme scaling from *fmax* to *fmin*— by our model with good accuracy. We also reverse engineer the power behavior of i7 core by measuring static energy dissipation, as well as dynamic energy consumed during the execution of programs and predicting how the power consumed by the processor is affected by frequency scaling.

Acknowledgments—This work is supported by the EU-FP7 ICT Projects, "A highly efficient adaptive multi-processor framework (HEAP)," Contract No. 247615, and "Embedded Reconfigurable Architecture (ERA)," Contract No. 249059.

8 References

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

[2] 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.

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

[4] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. Annual Int. Symposium on Microarchitecture, 2006.

[5] 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.

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

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

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

[9] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. Int. Conference on Computer Frontiers, 2010.

[10] G. Maglis, et al. Profile-based dynamic power voltage and frequency scaling for a multiple clock domain processor. Int. Conference on Computer Architecture, 2003.

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

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