Portable, Scalable, per-Core Power Estimation for Intelligent Resource Management

Abstract

Performance, power, and temperature are now all first-order design constraints. Balancing power efficiency, thermal constraints, and performance requires some means to convey data about real-time power consumption and temperature to intelligent resource managers. Resource managers can use this information to meet performance goals, maintain power budgets, and obey thermal constraints. Unfortunately, obtaining the required machine introspection is challenging. Most current chips provide no support for per-core power monitoring, and when support exists, it is not exposed to software. We present a methodology for deriving per-core power models using sampled performance counter values and temperature sensor readings. We develop application-independent models for four different (four- to eight-core) platforms, validate their accuracy, and show how they can be used to guide scheduling decisions in power-aware resource managers. Model overhead is negligible, and predictions exhibit 2.8%-5.2% median error on the NAS, SPEC OMP, and SPEC 2006 suites.

1 Introduction

Power and temperature have become first-order system design constraints. Together with performance, all three influence each other, and together they affect architectural and packaging choices. Power consumption characteristics further influence operating cost, reliability, battery lifetime, and device lifetime. Balancing power efficiency and thermal constraints with performance requires intelligent resource management. Achieving that balance, in turn, requires real-time power consumption and temperature information broken down according to resource, together with software and hardware that can leverage such information to enforce management policies.
One logical place to institute intelligent resource management with respect to power, performance, and temperature for chip multiprocessor (CMP) systems is at the level of individual cores. Measuring run-time power of a single core is problematic, though. Current chips do not support it. Power meters only report total consumption for everything behind a single power cable, and even if such aggregate data were sufficient, the use of meters becomes completely infeasible as machines scale up: coordinating output and feedback from thousands of meters would require a separate (super) computing system. Cycle-level system simulators provide in-depth information, but are extremely time consuming and prone to error. Power models implemented on top of the architectural abstractions in such simulators are inherently inaccurate [19], and are impossible to verify when attempting to assess new architectural designs. Hardware could be enhanced to measure the current and power draw of a CPU socket, but per-core measurement is difficult when cores share a power plane. Embedding measurement devices on-chip is usually infeasible. Even when measurement facilities exist — e.g., the Intel Core i7 [16] features per-core power monitoring at the chip-level — they are rarely exposed to the user. Indeed, power sensing, actuation, and management support is more often implemented at the blade level with a separate computer monitoring output [14, 21].

On the other hand, temperature sensors are now commonly available per core, and processor architectures generally support some number of performance monitoring counters (PMCs). We use data from these sensors and counters to generate accurate, per-core power estimation models. We create these models precisely because we lack direct information on power consumed.

Previous work uses PMCs to estimate power on uniprocessors and SMPs [18, 10, 24, 22]. Here we estimate power for both single-threaded and multithreaded programs on CMPs, observing median errors of 2.0% (2.6% mean), 2.8% (3.3% mean), 3.8% (5.0% mean), 4.4% (4.9% mean) for a quad-core Intel Q6600 CMP, an eight-core Intel E5430, a quad-core AMD Phenom, and an eight-core AMD Opteron 8212, respectively. We then use our online estimation to help manage workloads and frequencies dynamically.

This paper makes several contributions. We develop a model that uses performance monitoring counters to generate accurate per-core power estimates in real time, using no off-line profiling or tracing. The approach is
workload-independent, needs no knowledge of applications, and requires no application modification. We show the effectiveness of the model for driving power-aware scheduling and DFS: schedulers whose decisions are informed by these power models can enforce stricter power budgets, usually with little or no loss in performance. Finally, we demonstrate portability to four different CMP platforms. Studying sources of model error highlights the need for hardware support for power-aware resource management, such as fine-grained power sensors across the chip and more accurate temperature information. The proposed approach shows promise for balancing performance, power, and thermal requirements for platforms ranging from embedded real-time systems to consolidated data centers, and even to supercomputers.

2 Methodology

Building an analytic software model of per-resource power consumption requires some form of machine introspection. Any model must be based on knowledge of how and when a resource is being used. The resource on which we focus here is the processor core, and we leverage nearly ubiquitous performance monitoring counters (PMCs) to glean information about microarchitectural activity.

To build the model, we must 1) choose appropriate PMCs to reflect major activities consuming power, and 2) generate a mathematical model capturing relationships between chosen PMCs and measured system power. Other PMC-based power models use counters chosen a priori [22, 24]. Instead, we formalize the relationship between observed counter activity and measured system power with statistical analysis, choosing PMCs that correlate most strongly with measured power. We find that the best set of counters to be unique to each system. Having chosen sources of data to represent microarchitectural activity (and thus power draw), we sample those data while exercising the microarchitecture as fully as we can, and develop a linear regression model to fit collected data to measured power values. We derive power for the rest of the system external to the cores, and sum this to the per-core estimates to produce a verifiable system power estimate.
Note that hardware performance counters have limitations. For instance, most Intel platforms only support concurrent sampling of two counters. We multiplex counters, sampling two counters during each half of the time slice. A similar time-slicing strategy would be necessary were we to employ more counters than can be sampled concurrently on any architecture. Many counters report aggregate events at the chip level, and not per core, thus they not suited to our approach. In any architecture, the PC for the instruction triggering a counter overflow is often not the one reported. In some contexts, this sample skid can be problematic, but in the time-based sampling used to create our models, though, precision with respect to the instructions that cause counted events is unimportant.

2.1 Choosing PMCs

Current systems include a variety of event counters. We identify those most relevant to our purposes, rank them, evaluate our choices, and create a formal model mapping observed events to measured system power.

Identifying Candidates. To choose appropriate PMCs, we consider common events that represent the heavily used parts of a core’s microarchitecture. For instance, monitoring last-level cache misses allows us to track the most power-hungry memory hierarchy activity. Monitoring instructions retired along with their types allows us to follow power usage in the FPUs and integer units. Tracking total instructions retired gives information on overall performance and power. We expect out-of-order logic to contribute to power usage, as well. Even though no counter gives such information directly, monitoring resource stalls (stalls due to branches or full load/store queues, reorder buffers, and reservation stations) can provide insight. Stalls from issue logic may reduce power, but CPU stalls from full reservation stations, load-store queues, or reorder buffers may increase power, since the hardware attempts to extract instruction-level parallelism from the instruction stream. For example, if a fetched instruction stalls, the out-of-order logic tries to find another to execute, examining reservation stations to check each new instruction’s dependences and using more dynamic power. We separate available PMCs into the smallest set that covers these contributions to dynamic power, deriving four categories (a reasonable starting place, since our systems monitor two or four counters simultaneously): FP Units, Memory, Stalls, and Instructions Retired.
**Ranking Counters.** We create microbenchmarks to exercise different parts of the machine. We use no code from test benchmark suites, since the model must be application independent. We explore the space spanned by the four categories and attempt to cover common cases as well as extreme boundaries. The resulting counter values have large variations ranging from zero to several billion. For example, CPU-bound benchmarks have few cache misses, and integer benchmarks have few FP operations. The microbenchmarks have large for loops and case statements that branch to different code nests. We compile with no optimization to prevent redundant code removal, and run the microbenchmarks simultaneously on each core. All behavior should fall within the space defined by our four categories, and so the approach should work well on both current and future applications.

While running the microbenchmarks (once, at system startup), we collect power, and performance data for the PMCs in the four categories. We use a Watts Up Pro power meter to measure system power\(^1\) and pfm\textit{mon} to collect PMC data. We order the PMCs according to Spearman’s rank correlation to measure the relationship between each counter and power, and select the top PMCs from each category.

Even though we select event categories that attempt to capture different kinds of microarchitectural activity, we’re analyzing a cohesive hardware component for which all counters track inter-related events. This means that all counters chosen will necessarily contribute some redundant information. To assess this, we select several top-ranked PMCs in each category, and analyze the correlations among the PMCs themselves. If the top-ranked counters correlate strongly with other counters, we investigate counters that correlate less strongly with both power and other candidate counters, and evaluate accuracy of several alternate models. For the systems we study here, correlation of observed event rates and measured power is the strongest criterion for choosing PMCs: alternative models that try to leverage less redundant information (but perhaps less information \textit{in toto}) yield poorer models. In the (current) absence of a mathematical formula for evaluating the information redundancy versus the power contributions of a set of events, this sanity check helps ensure that we’re generating high quality models.

\(^1\)Note that we leave the meter plugged in to gauge model accuracy during normal operation, but when not testing accuracy, the meter is no longer required once the model is formed.
2.2 Forming the Machine Model

Having identified events that contribute significantly to consumed power, we create a formalism to map observed microarchitectural activity and measured temperatures to measured power draw. We re-run the microbenchmarks sampling just the chosen PMCs, collecting power and temperature data at each sampling interval. We normalize each time-sampled PMC value, $e_i$, to the elapsed cycle count to generate an event rate, $r_i$. We then map rise in core temperature, $T$, and the observed event rates, $r_i$, to core power, $P_{\text{core}}$, via a piece-wise model based on multiple linear regression, as in Equation 1. We apply linear weights to transformations of event rates, as in Equation 2.

$$
\hat{P}_{\text{core}} = \begin{cases} 
F_1(g_1(r_1), \ldots, g_n(r_n), T), & \text{if condition} \\
F_2(g_1(r_1), \ldots, g_n(r_n), T), & \text{else}
\end{cases}
$$

where $r_i = e_i / \text{(cycle count)}$

$$
F_n = p_0 + p_1 * g_1(r_1) + \ldots + p_n * g_n(r_n) + p_{n+1} * T
$$

We use transformations to increase linearity or to achieve constant variance in the regression equation (to make the model more amenable to linear regression and to improve estimation accuracy). Since Spearman’s rank correlation is independent of frequency distributions, transformations can be linear, inverse, logarithmic, exponential, or square root. We choose a piece-wise linear function because we observe significantly different behavior for low PMC values. This lets us retain the simplicity of linear regression and captures more detail in the core power function. We examine counter values plotted against power measurements to find opportunities for transformations, as well as to identify candidate locations for splitting data to form piece-wise models. For example, for the (hypothetical) data in Figure 1(a), neither a linear nor exponential transformation fits well. By breaking the data into parts, we create the better-fitting piece-wise combination in Figure 1(b). We use a least squares estimator to determine weights for function parameters, yielding the piece-wise model in Equation 3. The first equation estimates power when the counter is very low, and the second estimates power for the rest of the space.

$$
\hat{P}_{\text{core}} = \begin{cases} 
7.699 + 0.026 * \log(r_1) + 8.458 * r_2 + -3.642 * r_3 + 14.085 * r_4 + 0.183 * T, & r_1 < 10^{-6} \\
5.863 + 0.114 * \log(r_1) + 1.952 * r_2 + -1.684 * r_3 + 0.110 * \log(r_4) + 1.044 * T, & r_1 \geq 10^{-6}
\end{cases}
$$
2.3 Applying the Model

System power is based on measuring the power supply’s current draw from the power outlet when the machine is idle. When we cannot find published values for idle processor power, we sum power draw when powered off and power saved by turning off cdrom, floppy, and hard disk drives; removing all but one memory DIMM; and disconnecting fans. We subtract idle core power from idle system power to get uncore (baseline without processor) power. Change in the uncore power itself (due to DRAM or hard drive accesses, for instance) is included in the model predictions. Including temperature as a model input accounts for variation in uncore static power. Note that we always run in C0: experimenting with other power states is future work.

3 Experimental Results

We evaluate our models by estimating per-core power for the SPEC 2006 [27], SPEC-OMP [4, 26], and NAS [6] benchmark suites. We use gcc 4.2 to compile our benchmarks for a 64-bit architecture with optimization flags enabled, and run all benchmarks to completion on top of Linux kernel version 2.6.27. We use the pfmon utility from the perfmon2 library [13] to access hardware performance counters. Table 1 gives system details. Idle processor temperature is 36°C for both the Intel Q6600 and AMD Phenom platforms. Idle system power is 141.0W for the Intel Q6600, and 84.1W for the AMD Phenom. We subtract 38.0W [1] idle processor power to obtain an uncore
Table 1. Machine Configuration Parameters

<table>
<thead>
<tr>
<th>Category</th>
<th>Intel Q6600</th>
<th>Intel Xeon E5430</th>
<th>AMD Phenom 9500</th>
<th>AMD Opteron 8212</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores/Chips</td>
<td>4, dual dual-core</td>
<td>8, dual quad-core</td>
<td>4</td>
<td>8, quad dual-core</td>
</tr>
<tr>
<td>Frequency (GHz)</td>
<td>2.4</td>
<td>2.0, 2.66</td>
<td>1.1, 2.2</td>
<td>2.0</td>
</tr>
<tr>
<td>Process (nm)</td>
<td>65</td>
<td>45</td>
<td>65</td>
<td>90</td>
</tr>
<tr>
<td>L1 Instruction</td>
<td>32 KB 8-way</td>
<td>32 KB 8-way</td>
<td>64 KB 2-way</td>
<td>64 KB 2-way</td>
</tr>
<tr>
<td>L1 Data</td>
<td>32 KB 8-way</td>
<td>32 KB 8-way</td>
<td>64 KB 2-way</td>
<td>64 KB 2-way</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4 MB 16-way shared</td>
<td>6 MB 16-way shared</td>
<td>512 KB 8-way exclusive</td>
<td>1 TB 2-way exclusive</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>off-chip, 2 channel</td>
<td>off-chip, 4 channel</td>
<td>on-chip, 4 channel</td>
<td>on-chip, 2 channel</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4 GB DDR2-800</td>
<td>8 GB DDR2-800</td>
<td>4 GB DDR2-800</td>
<td>16 GB DDR2-667</td>
</tr>
<tr>
<td>Bus (MHz)</td>
<td>1066</td>
<td>1333</td>
<td>1100, 2200</td>
<td>1000</td>
</tr>
<tr>
<td>Max TDP (W)</td>
<td>105</td>
<td>85</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>2.6.27</td>
<td>2.6.27</td>
<td>2.6.25</td>
<td>2.6.31</td>
</tr>
</tbody>
</table>

Table 2. PMCs Selected for Each Platform

power of 103W for the Q6600, and subtract 20.1W [2] idle processor power to obtain an uncore power of 64.0W for the Phenom. Idle processor temperature for the Intel E5430 system is 45°C, and uncore power is 122.0W. Idle processor temperature for the AMD 8212 is 33°C, and uncore power is 90W. Table 2 lists chosen counters. We use the sensors utility from the lm-sensors library to obtain core temperatures, and we use a Watts Up Pro power meter [12] to gather power data. This meter is accurate to within 0.1W, and it updates once per second. We thus verify our power estimates at this one-second granularity.

We test our models for both multithreaded and single-threaded applications on our four platforms. We assess model error by comparing our system power estimates to power meter output (which our power meter limits to a one-second granularity). Then we incorporate the power model into a proof-of-concept, power-aware resource manager (a user-level meta-scheduler) designed to maintain a specified power envelope. The meta-scheduler manages processes non-invasively, requiring no modifications to the applications or OS. It does so by suspending/resuming processes and, where supported, applying DFS to alter core frequencies. Finally, we incorporate the model in a kernel scheduler, implementing a pseudo power sensor per core. The device does not exist in hardware, but is simulated by the power model module. Reads to the pseudo-device retrieve the power estimate
computed most recently. Taken together, our schedulers demonstrate the feasibility of using our models in power management at multiple levels of the software stack.

### 3.1 Computation Overhead

If computing the model itself is too expensive, its use becomes limited to course timeslices. The Linux kernel scheduler we use is specifically tailored to HPC applications, and so it introduces no overhead of its own. In most cases, it delivers better performance with better predictability and reduces OS noise [8, 9]. The pseudo per-core power sensor’s overhead depends on 1) the frequency with which the per-core power is updated, and 2) the complexity of the operations required to calculate the model. We calculate overhead by measuring execution time for applications running with a standard Linux scheduler and with our modified scheduler that computes the power model. We vary the sample period for reading the PMCs and temperature sensor (and computing the model) from one second to 10 msec. We time applications for five runs and take the average (variations among runs are in the noise). Table 3 gives measured times for the standard scheduler with and without computing model predictions.

### 3.2 Estimation Error

In our experiments, we run multithreaded benchmarks with one thread per core, and single-threaded benchmarks with an instance on each core. Data are calculated per core, and error is reported across all cores. Figure 2, Figure 3, and Figure 4 and Figure 5 show percentage error for the NAS, SPEC-OMP, and SPEC 2006 applications on all systems. Figure 6 through Figure 9 show standard deviation of error for each benchmark suite on the Intel

---

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>baseline (10msec)</th>
<th>model (10msec)</th>
<th>model (100msec)</th>
<th>model (1sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep.A serial</td>
<td>35.68</td>
<td>35.57</td>
<td>36.04</td>
<td>35.59</td>
</tr>
<tr>
<td>ep.A OMP</td>
<td>4.64</td>
<td>4.89</td>
<td>4.77</td>
<td>4.74</td>
</tr>
<tr>
<td>ep.A MPI</td>
<td>4.77</td>
<td>4.72</td>
<td>4.73</td>
<td>4.75</td>
</tr>
<tr>
<td>cg.A serial</td>
<td>5.82</td>
<td>5.83</td>
<td>5.83</td>
<td>5.83</td>
</tr>
<tr>
<td>cg.A OMP</td>
<td>1.95</td>
<td>1.95</td>
<td>1.95</td>
<td>1.95</td>
</tr>
<tr>
<td>cg.A MPI</td>
<td>2.19</td>
<td>2.20</td>
<td>2.20</td>
<td>2.20</td>
</tr>
<tr>
<td>ep.B serial</td>
<td>146.53</td>
<td>146.52</td>
<td>146.52</td>
<td>146.77</td>
</tr>
<tr>
<td>ep.B MPI</td>
<td>18.95</td>
<td>19.41</td>
<td>19.12</td>
<td>19.18</td>
</tr>
<tr>
<td>cg.B serial</td>
<td>559.58</td>
<td>560.50</td>
<td>560.11</td>
<td>560.19</td>
</tr>
<tr>
<td>cg.B OMP</td>
<td>91.29</td>
<td>92.64</td>
<td>96.90</td>
<td>89.92</td>
</tr>
<tr>
<td>cg.B MPI</td>
<td>79.11</td>
<td>79.18</td>
<td>79.18</td>
<td>79.05</td>
</tr>
</tbody>
</table>

**Table 3.** Scheduler Benchmark Times (average of five runs)
Figure 2. Median Estimation Error for Intel Q6600 system

Figure 3. Median Estimation Error for Intel E5430 system

Figure 4. Median Estimation Error for the AMD Phenom 9500

Figure 5. Median Estimation Error for the AMD Opteron 8212
E5430, the AMD Phenom, and the AMD Opteron platforms. The occasional high standard deviations illustrate one problem with our current infrastructure: instantaneous power measurements once per second do not tell us what has been happening to the performance counters since the last meter reading. Our Linux kernel scheduler will allow us to experiment with much finer timeslices.

Estimation error ranges from 0.3% (leslie3d) to 7.1% (bzip2) for the Intel Q6600 system, from 0.3% (ua) to 7.0% (hmmer) for the Intel E5430 system, from 1.02% (bt) to 9.3% (xalancbmk) for the AMD Phenom system, and from 1.0% (bt.B) to 10.7% (soplex) for the AMD Opteron 8212 system. Our model exhibits median error of 1.8%, 5.1%, and 3.0% for SPEC-OMP, SPEC 2006, and NAS, respectively, on the Q6600. Median errors per suite are 2.8%, 3.5%, and 3.9% on the E5430, 5.2%, 3.5%, and 4.5% on the Phenom, and 3.4%, 4.8%, and 2.5% on the 8212. On the Intel Q6600, only five (out of 45) applications exhibit median error exceeding 5%; on the Intel E5430, only six exhibit error exceeding 5%; and on the AMD Phenom, eighteen exhibit error exceeding 5%. On the AMD Opteron 8212, thirteen exhibit error over 5%. This error is not excessive, but lower is better: identifying causes of error and developing model refinements are ongoing work. For instance, the Opteron 8212 temperature sensor data appear inaccurate: we measure different idle temperatures for different cores. We achieve good results despite the inaccuracies by underestimating idle temperature and using this value consistently for model training, testing, and dynamic calculation. Prediction errors are not clustered, but are spread throughout application execution. Monitoring performance counters at a finer granularity reveals that our meter’s one-second resolution for delivering instantaneous power measurements prevents capturing counter activity between samples.

Figure 10 shows model coverage via Cumulative Distribution Function (CDF) plots for the suites. On the Q6600, 85% of estimates have less than 5% error, and 98% have less than 10%. On the E5430, 73% have less than 5% error, and 99% less than 10%. On the Phenom, 59% have less than 5% error, and 92% less than 10%. On the 8212, 37% have less than 5% error, and 76% have less than 10%. Most estimates exhibit very small error.

Figure 11 through Figure 14 compare measured to estimated power for our experimental platforms. Values are calculated as the mean of power over an application’s execution. These data show that our models do not
Figure 6. Standard Deviation of Error for Intel Q6600 system

Figure 7. Standard Deviation of Error for Intel E5430 system

Figure 8. Standard Deviation of Error for the AMD Phenom 9500

Figure 9. Standard Deviation of Error for the AMD Opteron 8212

Figure 10. Cumulative Distribution Function (CDF) Plots Showing Fraction of Space Predicted (y axis) under a Given Error (x axis) for Each System
consistently under- or over-estimate power, tracking it well for each suite.

3.3 Live Power Management

We use per-process power estimates to enforce a given power budget for multiprogrammed workloads on each CMP. Table 4 presents three sets of multiprogrammed workloads exhibiting varying levels of *computational intensity*, which we define as the ratio of instructions retired to cache misses. The first set contains benchmarks exhibiting the highest intensity, the second, medium intensity, and the third, lowest intensity.

We experiment with two example policies for our meta-scheduler. The policies themselves are not particularly interesting, but they try to optimize for different goals, and together they provide a proof of concept with respect
to the use of our power models in resource management. When the power envelope is breached on a system supporting DFS, the meta-scheduler reduces the frequency of the core running the process selected by the policy. When DFS is unavailable or exhausted, the meta-scheduler suspends the selected process. When there is power slack, the meta-scheduler resumes the process selected by the policy, or if all processes are running and DFS is supported, it increases the frequency of the core running the selected process. The max inst/watt policy attempts to achieve the most energy-efficiency under a given power envelope by selecting the process with the least instructions committed per watt of power consumed when the power budget is exceeded, and the one with the most instructions committed per watt when there is power slack. The per-core fair policy tries to allocate an equitable share of available power per core. When the envelope is breached, the policy selects the process with highest average consumed power. When there is power slack, the policy selects the process with lowest average consumed power.

We perform two sets of experiments. For the first set, we use the 2.2 GHz AMD Phenom and 2.4 GHz Intel Q6600 systems. We selectively suspend/resume processes to remain within the system power envelope. For the

<table>
<thead>
<tr>
<th>Benchmark Set</th>
<th>Intel Q6600 and AMD Phenom</th>
<th>Intel E5430</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU bound</td>
<td>ep, game, namd, povray</td>
<td>calculix, ep, game, gromacs, h264ref, namd, perlbench, povray</td>
</tr>
<tr>
<td>Moderate</td>
<td>art, lu, wupwise, xalancbmk</td>
<td>bwaves, cactusADM, fma3d, gcc, lesleed, sp, ua, xalancbmk</td>
</tr>
<tr>
<td>Memory bound</td>
<td>astar, mcf, milc, spopx</td>
<td>applu, astar, bpm, mcf, milc, omnetpp, spopx, xalancbmk</td>
</tr>
</tbody>
</table>

Table 4. Experimental Multiprogrammed Workloads
The power manager can thus better implement policy decisions by estimating power for either frequency. Since the E5430 platform supports DFS per (quad-core) chip, and not per core, we independently vary frequencies among the two chips. If all core frequencies have been scaled down and the power envelope is still breached, we suspend processes to reduce power. We compare against runtimes with no enforced power envelope.

Figure 15 and Figure 16 show examples of the two policies with different power envelopes for the Intel Q6600 and AMD Phenom systems. Measured and predicted power correlate well. We follow the power envelope, and we
do so entirely on the basis of our estimation-based power manager. This obviates the need for a power meter, and is
a useful tool for software control of total system power. Figure 17 and Figure 18 show normalized runtimes for the
two sample policies and all power envelopes on both systems. This meta-scheduler only suspends/resumes pro-
cesses. Performance sometimes improves for the 95% and 90% power envelopes, since with no power envelope,
all processes compete for cache and memory bandwidth. When power envelopes are enforced, the meta-scheduler
throttles processes to conserve power, making more cache and memory bandwidth available to the executing pro-
cesses. This can speed application execution. This effect disappears for the 85% envelope: even though there is
less competition, processes cannot execute at maximum speed, or they will breach the power envelope.

For the second set of experiments we vary power usage by choosing between frequencies of 1.1 and 2.2 GHz
on the AMD Phenom system and between 2.0 and 2.67 GHz on the Intel 5430 system. If DFS is insufficient, the
meta-scheduler suspends processes. Figure 19 and Figure 20 show normalized runtimes for the two policies and
all power envelopes on the AMD Phenom and E5430 systems. Note the different scales of the $y$ axes: perfor-
mance losses for shrinking power envelopes on the Phenom are much smaller with DFS. For that machine, the
performance-oriented policy maintains performance in all cases. On the Intel E5430, we lose under 4% perfor-
mance for all but the CPU-bound workload under the strictest power envelope. The performance differences for
the policies shows that DFS alone does not maintain the power envelope (except in the case of the memory bound
workloads), and that the policies obviously suspend different processes for the CPU-bound and moderate work-
loads. Memory-bound workloads benefit most from the policy that tries to allocate power equally among cores.
Likewise, those same workloads suffer little or no performance loss with stricter power budgets when DFS and
scheduling are used to enforce power budgets. Lower frequency slows computation, but since it does not impact
memory performance, the non-CPU-bound workloads suffer little (and sometimes speed up).
4 Related Work

Much previous work leverages performance counters for power estimation. Many of these do not report model error, and none presents results as extensively as we do here. Joseph and Martonosi [18] use PMCs to estimate power in a simulator (SimpleScalar [5]) and on real hardware (a Pentium Pro). Their hardware supports two PMCs, while they require twelve. They perform multiple benchmark runs to collect data, forming the model offline. Multiplexing twelve PMCs would require program behavior to be relatively constant across the entire sampling interval. They cannot estimate power for 24% of the chip, and assume peak power for those structures.

Contreras and Martonosi [10] use five PMCs to estimate power for different frequencies on an XScale system (with an in-order uniprocessor). Like Joseph and Martonosi, they gather data from multiple benchmark runs. They derive power weights for frequency-voltage pairs, and form a parameterized linear model. Their model exhibits low percentage error, like ours, but they test on seven applications, and their methodology is only demonstrated for a single platform. This methodology, like that of Joseph and Martonosi, is not intended for on-line use.
Economou et al. [11] use PMCs to predict power on a blade server. They profile the system using application-independent microbenchmarks. The resulting model estimates power for the CPU, memory, and hard drive with 10% average error. Wu et al. [29] also use microbenchmarks to develop a model using a variable number of PMCs to estimate active power of Pentium 4 functional units. They measure CPU power with a clamp-on ammeter. Such invasive approaches are not applicable to systems deployed in the field, nor scalable to HPC clusters. Rajamani et al. [24] develop online power and performance models on a Pentium M system. They track a small number of PMCs (chosen \textit{a priori}), and develop their model using existing kernels (as opposed to microbenchmarks specifically designed to exercise the counters). They present two power-management strategies (one targeting performance and the other power savings) that leverage their models for dynamic p-state control. Their policies are effective in meeting power and performance requirements. No percentage errors are reported for model estimates.

Merkel and Bellosa [22] use PMCs to estimate power per processor in an eight-way symmetric multiprocessing (SMP) system, shuffling processes to reduce overheating of individual processors. They do not state which PMCs they use, nor how they are chosen. Their goal is not to reduce power, but to distribute it. Their estimation method suffers less than 10% error in a Linux kernel implementation.

Lee and Brooks [20] predict power via statistical inference models. They build correlations based on hardware parameters, using the most significant parameters to train their model. They profile their design space \textit{a priori}, and then estimate power on random samples. This methodology requires training on the same applications for which they want to estimate power, and so their approach depends on having already sampled all applications of interest.

Isci et al. [17] analyze power management policies for a given power budget, performing experiments on the Turandot [23] simulator. They assume on-core current sensors to obtain per-core power, while we propose a technique to model per-core power without additional hardware. They leverage per-core and domain-wide DVFS to retain performance while remaining within a power budget. While their work involves core adaptation policies, we explore both suspending threads as well as DVFS, to make our approach applicable to more systems.

Instead of applying DVFS per core, Rangan et al. [25] study thread placement and migration among independent
voltage and frequency domains. This *thread motion* permits much finer-grain control over power management and delivers better performance than traditional DVFS for a given power budget. Grochowski et al. [15] study four methods to control energy per instruction (EPI), finding that a combination of VFS and asymmetric cores offers the most promising approach to balancing latency and throughput in an energy-constrained environment. Annavaram et al. [3] leverage this work to throttle EPI by scheduling multithreaded tasks on an asymmetric CMP, using more EPI in periods of low parallelism to make better use of a fixed power budget.

Banikazemi et al. [7] present a power-aware meta-scheduler. They exploit the built-in power monitoring hardware on an IBM blade server with eight Intel Xeon 5345 cores (four dual-core chips). They use PMC data to calculate cache occupancy, miss ratios, and CPI, deriving a performance model from which they estimate system power consumption.

5 Conclusions and Future Work

We derive analytic, piece-wise linear power models mapping PMC values and temperature readings to power, and demonstrate their accuracy for the SPEC 2006, SPEC-OMP and NAS benchmark suites. We write microbenchmarks to exercise the PMCs in order to characterize the machine, and run those microbenchmarks with a power meter plugged in to generate data for building the models. We select the PMCs that correlate most strongly with measured power for our regression models. Because they are built on microbenchmark data, and not actual workload data, the resulting models are application independent. We apply the models to 45 benchmarks (including multithreaded applications) on four CMPs containing dual- or quad-core chips totaling four or eight cores. We omit data for Core 2 Duo and Core i7 due to space considerations: all trends are similar to those presented here. Previous work focuses largely on uniprocessor power models (that do not incorporate temperature), and each methodology targets a single architecture. In spite of our generality, estimation errors are consistently low, and the approach appears portable. When using the models to estimate power during application execution, we observe median errors between 2.8% (3.3% mean) and 4.4% (4.9% mean).
We leverage our power model to perform live power management on CMP systems with and without VFS. We suspend and resume processes based on per-core power usage, ensuring that a given power envelope is not breached. Where supported, we also scale frequency for cores to remain under the power envelope, but VFS alone is insufficient for enforcing the budgets in our experiments. Our power managers enforce power budgets with little or no performance loss.

Estimating per-core power is challenging, since some resources are shared among cores (e.g., caches, the DRAM memory controller and off-chip memory). Current machines only permit sampling a few PMCs simultaneously per core, which necessitates multiplexing the counters for models requiring more. This affects the size of the sampling interval for applying the model, something we have just begun to study. We have incorporated the model into an HPC Linux kernel scheduler, and are in the process of experimenting with different scheduling policies. Having a unified infrastructure permits a fair comparison of power-aware thread scheduling with techniques like thread motion. Tracking PMC values at finer timeslices than that for which our meter can measure power emphasizes the need for fine-grain power sensors on chip: our model is sufficient to be useful, but much finer resource management would be enabled by making more information available from hardware.

As numbers of cores and power demands continue to grow, efficient computing requires better methods for managing individual resources. The per-core power estimation methodology presented here extends previously published models in both breadth and depth, and represents a promising tool for helping meet those challenges, both by providing useful information to resource managers and by highlighting opportunities for improving hardware support for energy-aware resource management.

References


