Performance and Power Modeling in a Multi-Programmed Multi-Core Environment

Xi Chen
EECS Department
University of Michigan
Ann Arbor, MI 48105
chexi@umich.edu

Chi Xu
ECE Department
University of Minnesota
Minneapolis, MN 55455
xuchi@umn.edu

Robert P. Dick
EECS Department
University of Michigan
Ann Arbor, MI 48105
dickrp@eecs.umich.edu

Zhuoqing Morley Mao
EECS Department
University of Michigan
Ann Arbor, MI 48105
zmao@eecs.umich.edu

ABSTRACT

This paper describes a fast, automated technique for accurate on-line estimation of the performance and power consumption of interacting processes in a multi-programmed, multi-core environment. The proposed technique does not require modifying hardware or applications. The performance model uses reuse distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each process to predict throughput. The system-level power model is derived using multi-variable linear regression, accounting for cache contention. Both models are validated on multiple real multi-core systems using SPEC CPU2000 benchmarks; their performance and power estimates are within 3.5% of measured values on average. We explain how to integrate the two models for power estimation during process assignment, helpful for power-aware assignment.

Categories and Subject Descriptors

C.0 [**General**]: Modeling of computer architecture

General Terms

Experimentation

Keywords

performance modeling, power modeling, assignment

1. INTRODUCTION AND MOTIVATION

Performance and power modeling in a multi-programmed single-core environment is challenging due to issues such as time sharing among processes. The on-going move to chip multiprocessors (CMPs) permits sharing the last-level cache among cores on the same die but this aggravates the cache contention problem: processes running simultaneously on cache-sharing cores contend for the limited space in the last-level cache, impacting performance and power consumption, which further complicates the modeling problem. Accurately modeling the performance and power consumption in a multi-programmed multi-core environment is necessary for design-time architectural optimization and run-time dynamic resource management [3, 7].

Performance and power modeling in a multi-programmed multi-core environment presents several challenges: (1) the models should be easy to construct without modifications to existing software or hardware. Exhaustive off-line simulation of all process combinations is computationally intractable and thus should be avoided; (2) the models should handle time sharing among processes on the same core and resource contention among processes on cache-sharing cores; and (3) to be useful in on-line process assignment, the models must estimate power and throughput before processes are assigned. To the best of our knowledge, no existing performance and/or power models satisfy the requirements mentioned above.

This paper makes the following contributions: (1) we propose a modeling framework that generates fast, accurate, on-line estimates of performance and power consumption for any process-to-core mapping during runtime; (2) the system-level power model can handle time sharing among processes on the same core and cache contention among processes on cache-sharing cores; (3) this is the first work to estimate the processor power for any tentative assignment without runtime information by integrating the performance model and the power model; and (4) our models are general enough to accommodate heterogeneous tasks and processors. Both models have been validated on different machines with different architectures and nominal power consumptions. Note that although constructing a performance model requires profiling each process of interest, this does not limit the generality of our approach because profiling can be done on-line. When a new application makes up a significant percentage of the workload, we force it to run alone on an idle machine and record profiling information. Therefore, the approach can be used (in different ways) for both embedded and general-purpose computing systems.

2. RELATED WORK

This paper builds upon previous work on performance and power modeling.

Researchers considered addressing the performance pre-

This work was supported in part by NSF under awards CCF-0702761 and CNS-0347941 and in part by SRC under award 2007-HJ-1593.

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

diction problem with the assistance of offline simulation [1] or modifications to the existing system [9]. However, these techniques are either time-consuming or require substantial changes to the hardware or operating system. Other researchers address the performance prediction problem analytically. Chandra et al. proposed three analytical models based on the L2 reuse distance or circular sequence profile of each thread to predict inter-thread cache contention on a CMP system, assuming no data dependencies exist among threads [4]. Their work is the closest to ours. However, their models require a priori knowledge of the L2 cache access frequency of a process in the steady state when running concurrently with other processes. We can think of no practical way to obtain this information without running or simulating all potential combinations of concurrent cache-sharing processes. Our work builds upon the performance modeling technique proposed by Xu et al. [11], which uses processor performance counter data to predict the impact of cache contention on cache partitioning and performance. However, this prior work was designed for performance prediction and did not consider power modeling and estimation.

Researchers have also developed simulation-based power models [3]. However, such models impose significant performance overhead and are therefore inappropriate to use during runtime. Other researchers have proposed performance-counter-based power models for on-line power estimation [6, 7]. However, such models only estimate the power consumption of a single application; it is not straightforward to extend them for power estimation in a multi-programmed, multi-core environment. Singh et al. proposed a performance counter based power model in a multi-programmed CMP environment [8]. This work is related to ours. However, their power model construction process is ad hoc and requires the user manually tune the model parameters and fitting functions. In addition, their power model cannot handle time sharing among processes on the same core. In contrast, the model building process for our power model can be fully automated. As demonstrated in Section 6, it can handle time sharing among processes and applies to CMP systems with different architectures without any changes to the model construction process.

3. PERFORMANCE MODELING

In this section, we first formulate the performance modeling problem. We then give details on how to derive the non-linear equilibrium equations for effective cache size prediction. Finally, we describe the automated profiling process for performance prediction.

3.1 Problem Formulation and Assumptions

The problem of performance prediction in the presence of cache contention can be formulated as follows: given k processes assigned to cores sharing the same A-way set-associative last-level cache, predict the steady-state cache size occupied by each process during concurrent execution. Solving this problem is helpful for process assignment and migration in a CMP environment. However, accurate prediction of process performance is challenging due to the exponential number of possible process-to-core mappings.

In this paper, we consider a k-core processor with an L2 cache being the last-level on-chip cache. In the rest of paper, we refer to L2 cache simply as "cache" whenever this does not introduce ambiguity. We assume no hardware prefetching. Hardware prefetching complicates the model by predictively fetching cache lines based on access patterns. The model might therefore be inaccurate for systems using prefetching. However, we argue that prefetching is of limited value on CMPs with constrained processor–memory bandwidth. We evaluated 10 SPEC CPU2000 benchmarks to determine the performance impact of hardware prefeteching. Our experimental results indicate the average performance improvement was 3.25%, and only *equake* benefitted significantly. We also make the following assumptions: (1) the cache uses an LRU replacement policy and (2) processes are single-phased. In the case of multiple non-repeating phases with distinct memory access patterns, non-repeating phases should be modeled separately. Although these two assumptions simplify model design and explanation, we will later experimentally evaluate the proposed models when many of the assumptions are violated.

When multiple processes share a cache, contention may occur. We define the number of ways occupied by process i in a set, denoted as S_i, as the *effective cache size* associated with process i. Therefore,

$$\sum_{i=1}^{k} S_i = A, \tag{1}$$

where k is the total number of processes sharing the cache.

We define the *reuse distance*, R_i, of cache line i as the number of distinct cache lines within the same set accessed between two consecutive accesses to line i. A *reuse distance histogram* represents the distribution of cache line reuse distances for an entire shared cache. For process i with an effective cache size of S, all accesses to the cache lines with a reuse distance larger than S result in cache misses. Hence, the probability of a cache access resulting in a miss for a process with an effective cache size of S_i, i.e., the misses per access (MPA), can be expressed as follows.

$$\mathrm{MPA}_i(S_i) = \int_{S_i}^{\infty} \mathrm{hist}_i(x)\, dx. \tag{2}$$

We also experimentally determined that SPI, number of seconds per instruction, can be expressed as a linear function of MPA,

$$\mathrm{SPI} = \alpha \cdot \mathrm{MPA} + \beta, \tag{3}$$

where α and β are parameters that can be obtained during offline characterization. This observation is re-affirmed by Choi et al. [5].

3.2 Estimating Effective Size After n Accesses

In this section, we use the reuse distance histogram of a process to derive its effective cache size. To simplify explanation, we will for the moment assume that the cache is initially empty. This assumption will later be relaxed. Given that $P_{i,n}$ is the probability of having an effective cache size of i after n consecutive cache accesses, the following recursive equation can be derived:

$$P_{i,n} = P_{i,n-1}\cdot(1-\mathrm{MPA}(i))+P_{i-1,n-1}\cdot\mathrm{MPA}(i-1), 1 < i \leq n. \tag{4}$$

This can be explained as follows. The fact that n cache accesses result in an effective cache size of i can only be the result of the following two scenarios: (1) the first $n-1$ cache accesses lead to an effective cache size of i and the nth access results in a cache hit. The probability of this scenario, $P(A)$, is thus $P_{i,n-1} \cdot (1 - \mathrm{MPA}(i))$; (2) the first $n-1$ cache accesses lead to an effective cache size of $i-1$ and the nth access causes a cache miss. The probability of this scenario, $P(B)$, is thus $P_{i-1,n-1} \cdot \mathrm{MPA}(i-1)$. Since $P_{i,n} = P(A) + P(B)$, we can derive Equation 4. Note that $P_{1,1} = 1$ because the first cache access will always occupy a cache line. Assuming the process reaches its steady state after n accesses, let $G(n)$ be process k's effective cache size, we have

$$G(n) = \sum_{i=1}^{n} (P_{i,n} \times i) \tag{5}$$

3.3 Equilibrium Condition

Given a cache with an LRU-like replacement policy, it is reasonable to assume that at time t, we can always find a duration T such that data accessed before time $t-T$ have been evicted and data accessed during $[t-T, t]$ are preserved in the cache. Since none of these accesses will evict any data lines accessed during $[t-T, t]$, it is as if the data were written to an empty cache with no cache misses during $[t-T, t]$, which indicates Equation 4 and Equation 5 hold. Hence, the effective cache size of process i, denoted as S_i, can be written as $G_i(\text{APS}_i \cdot T)$. Conversely, APS_i can be expressed as $G_i^{-1}(S_i)/T$. From Equation 3, we can derive an expression for APS_i:

$$\text{APS}_i = G_i^{-1}(S_i)/T = \text{API}_i/(\alpha_i \text{MPA}_i(S_i) + \beta_i). \quad (6)$$

Note that Equation 6 holds for any process i, where $i = 1, 2, \cdots, k$, given that k is the total number of processes. Therefore, we have

$$\frac{G_1^{-1}(S_1)}{G_i^{-1}(S_i)} - \frac{\text{API}_1 \cdot (\alpha_i \text{MPA}_i(S_i) + \beta_i)}{\text{API}_i \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)} = 0, \forall_{i=1}^{k} \quad (7)$$

where $G^{-1}(S_i)$ and $\text{MPA}(S_i)$ are application-dependent nonlinear functions of S_i. Combined with Equation 1, we have k equations that are independent of each other. Newton–Raphson iteration can therefore be used to solve for each $S_i, 1 \le i \le k$.

3.4 Automated Performance Profiling

In this section, we describe a fast approach to extract reuse distance histograms of processes using hardware performance counters (HPCs). To obtain the reuse histogram information of each process, we assign the process concurrently with a carefully designed benchmark with configurable cache contention characteristics, which we will refer to as a *stressmark*. Based on the information collected from HPCs and Equation 7, we can calculate the reuse distance histogram.

Consider a process (denoted as B) and the stressmark running on two cores sharing an A-way last-level cache. We assume if the stressmark occupies i ways in a cache set, the concurrently running process, B, will occupy $A-i$ ways. We then obtain the reuse distance histogram of B as follows. Run the stressmark along with B multiple times. In the ith run, tune the parameters in the stressmark to change its effective cache size, denoted as $S_{\text{stress},i}$. Record B's MPA in each run, denoted as $\text{MPA}_{B,i}$, where $i = 1, 2, \cdots, A$. Given that $S_{B,i}$ is process B's effective cache size in the ith run, considering the ith and the $(i+1)$st runs, Equation 2 indicates

$$MPA_{B,j} = \int_{S_{B,j}}^{\infty} hist_B(x)dx, \; j = i, i+1.$$

Hence, we can estimate the probability of process B having an effective cache size of $S_{B,i}$ as

$$hist_B(S_{B,i}) \approx MPA_{B,i+1} - MPA_{B,i}. \quad (8)$$

By varying $S_{B,i}$ from 1 to A, we can estimate the probability at each effective cache size, thus allowing us to construct the reuse distance histogram. In reality, we tune $S_{\text{stress},i}$ to control $S_{B,i}$.

Process characterization can be automated as follows. We first run the stressmark along with the process A times, varying the effective cache size. After A runs, API is recorded, α and β in Equation 3 can be estimated using linear regression, and the reuse distance histogram can be estimated using Equation 8. These four parameters form the *feature vector* of a process. Given the feature vectors of several processes, we can predict the effective cache size of each process when they run on cores sharing the same (last-level) cache, which in turn can be translated to SPIs through Equation 2 and Equation 3. Hence, given k processes to be assigned to k cores, we only need to obtain k feature vectors ($\mathcal{O}(k)$ complexity) to predict the performance of any subset of the k processes for assignment ($2^k - 1$ combinations).

4. POWER MODELING

In this section, we first formulate the power modeling problem. We then explain the model construction process. Finally, we describe how we handle time sharing among processes sharing cores and cache contention among processes running on multiple cores.

4.1 Problem Formulation

The power modeling problem in a multi-programmed multi-core environment can be formulated as follows: given k processes running on N cores with some of the cores having multiple processes and some of them being idle, estimate the core and processor power consumption during concurrent execution.

It is natural to decompose core power consumption into idle power consumption and the active power consumptions of individual architectural blocks. Given that there are M components in a system, the total power consumption is $P = P_{\text{idle}} + \sum_{i=1}^{M} P_i$, in which P_{idle} is the idle power consumption when no process is actively using the core and P_i is the power consumption of component i. In order to make online estimates of P_i, we again use HPCs: by carefully choosing the HPC-detected hardware events monitored, we can map an event rate, i.e., number of events per second, to the power consumption of the corresponding architectural block. We first choose the HPC event rates that are most correlated to core power consumption. We omit the details here due to space limitations. The top 5 event rates with the highest correlation coefficients are L1RPS, L2RPS, L2MPS, BRPS, and FPPS, which represent the number of L1 data cache references per second, number of L2 cache references per second, number of L2 cache misses per second, number of floating point instructions retired per second, and number of branch instructions retired per second, respectively.

It remains unclear how to map the event rates to the corresponding component power: the power consumption of a component may be nonlinearly dependent on the event rate associated with it. We first wrote a micro-benchmark with 6 phases, each of which lasts 80 s. In the first phase, the core idle power is recorded, whereas one of the aforementioned 5 architectural blocks are explicitly accessed in each of the following 5 phases. Note that the access frequency is the highest at the start of a phase and reduced to a lower level every 10 s, i.e., there are 8 different access frequencies for one component in one phase. We then use 8 SPEC CPU2000 benchmarks (see Section 6) and the micro-benchmark for model construction. Given an N-core processor, we run N instances of one benchmark on N cores (one instance per core) and gather the HPC values along with the processor power throughout the execution, assuming each core has the same power and HPC values. We then evaluate the modeling results based on two different algorithms, the multi-variable linear regression (MVLR) algorithm and a three-layer sigmoid activation function neural network (NN). Experimental results indicate that the MVLR-based model achieves an accuracy of 96.2% while the NN-based model reaches an accuracy of 96.8%. Given an accuracy comparable to NN-based model and the simplicity in model construction and evaluation, MVLR-based model is chosen. Hence, the core power P_{core} can be expressed as

$$\begin{aligned} P_{\text{core}} \;=\; & P_{\text{idle}} + c_1 \cdot \text{L1RPS} + c_2 \cdot \text{L2RPS} + c_3 \cdot \text{L2MPS} + \\ & c_4 \cdot \text{BRPS} + c_5 \cdot \text{FPPS}, \end{aligned} \quad (9)$$

where P_{idle} and c_1 through c_5 are coefficients determined from MVLR.

4.2 Handling Context Switching and Cache Contention

The proposed power model can accurately estimate the core power consumption when a single process is running. However, there are usually multiple processes running on the same core in a multi-programmed environment, limiting the usability of the power model. We define *process power consumption* as the core power consumption when this process is running. Since we assume there are no data dependencies among processes, the major interactions among processes on the same core are contention for resources such as cache. We experimentally determined the average amount of time required to fill the cache after a context switch is only 1% of the timeslice length given a 20 ms timeslice, which indicates the impact of context switches on performance and power is negligible. Therefore, the core power consumption is the linear weighted sum of all process power consumptions with the timeslice length of each process being its weight. In reality, we make the simplifying assumption that every process has the same weight. Hence, assuming there are k processes running on the single core with process i's power consumption being P_i, the core power consumption is simply $P_{\text{core}} = \frac{1}{k}\sum_{i=1}^{k} P_i$.

We now define the *processor power consumption* as the sum of all core power consumptions in a multi-core multi-programmed environment, in which cache contention problem becomes more severe. On one hand, increased cache contention leads to lower processor power consumption because c_3 is negative in Equation 9. On the other hand, increased resource utilization implies higher processor power consumption. The amount of increase in processor power consumption depends on the balance between the two factors. This is consistent with our experimental results (see Section 6). Therefore, the proposed power model can handle the multi-core environment without any modifications. If there is more than one process per core, given core 1 through core N share the last-level cache and S_i is the set of processes running on core i, the average power consumption of these cores $P_{\text{core-set}}$ can be calculated as

$$P_{\text{core-set}} = \frac{\sum_{p_1 \in S_1} \cdots \sum_{p_N \in S_N} P(p_1, p_2, \cdots, p_n)}{\prod_{i=1}^{N} |S_i|}, \quad (10)$$

where $P(p_1, p_2, \cdots, p_n)$ is the sum of power consumptions of core 1 through core N when processes p_1, p_2, $\cdots$, p_n run simultaneously.

5. COMBINING PERFORMANCE AND POWER MODELS

In this section we describe how to combine the proposed performance and power models for use in optimization. One such application is power-aware assignment. More specifically, if we can accurately estimate the processor power consumption for each tentative assignment decision, we can choose the one that optimizes power or energy usage. However, such power estimation is usually impossible because the HPC values needed for power estimation are unknown until the processes are assigned. Nonetheless, by integrating the performance model and the power model, we are able to estimate the process power consumption for each assignment, as explained below.

Given the power model in Equation 9, we can decompose

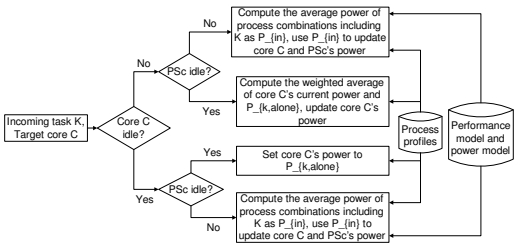

Figure 1: Algorithm for power estimation for process assignment.

the process power P_{process} into two parts:

$$\begin{aligned} P_1 &= P_{\text{idle}} + (c_1 \cdot \text{L1RPI} + c_2 \cdot \text{L2RPI} + \\ &\quad c_4 \cdot \text{BRPI} + c_5 \cdot \text{FPPI})/\text{SPI}, \\ P_2 &= c_3 \cdot \text{L2MPS} = c_3 \cdot \text{L2MPR} \cdot \text{L2RPI}/\text{SPI}, \text{and} \\ P_{\text{process}} &= P_1 + P_2. \end{aligned}$$

Here, P_{idle} is the power consumption of an idle core, L1RPI represents the number of L1 data cache accesses per instruction, L2RPI represents the number of L2 cache references per instruction, BRPI represents the number of branches per instruction, FPPI represents the number of floating point instructions retired per instruction, and L2MPR represents the number of L2 cache misses per L2 cache reference. We define a *instruction-related event rate* as the number of events per instruction. L1RPI, L2RPI, BRPI, and FPPI in P_1 are process properties: given the same input data, these instruction-related event rates are fixed and not affected by the execution of other processes. Therefore, the impact of cache contention is only reflected in the change of SPI. However, P_2 is not only influenced by SPI but also L2MPR. Fortunately, both SPI and L2MPR can be determined by the performance model given enough profiling information, as explained in Section 3. Hence, if we record the instruction-related event rates during profiling for each process and use performance model in Section 3 to predict SPI and L2MPR whenever cache contention exists, we can estimate P_1, P_2, and thus the process power.

We first assume the performance and power model are built as described in Section 3 and Section 4. We also assume for each process i, the profiling vector PF_i, i.e., ($P_{i,\text{alone}}$, L1RPI_i, L2RPI_i, BRPI_i, FPPI_i) is recorded during profiling. Note that $P_{i,\text{alone}}$ represents process i's average power consumption when it runs alone with no other active processes. Figure 1 illustrates how to combine the performance model, power model, and process profiles for power estimation during assignment. Suppose we want to evaluate the resulting power consumption by assigning process K to core C. We denote the set of cores that share the last-level cache with core C as core C's partner set PS_C. Depending on the states of core C and PS_C, there are four different outcomes: (1) both C and PS_C are idle, (2) C is busy and PS_C is idle, (3) C is idle and PS_C is busy, and (4) both C and PS_C are busy. We only analyze scenario (1) and scenario (4) since scenarios (2) and (3) are special cases of scenario (4). In scenario (1), we set core C's power consumption to $P_{K,alone}$, fetched from profiling vector PF_K. The processor power consumption is also increased by $P_{K,alone}$. In scenario (4), we assume there are N cores in PS_C numbered from 1 to N, among which core 1 through core m have processes running on them and core $m+1$ through core N are idle. For convenience, we use S_i to represent the set of processes running on core i. We define a *process combination* as an ordered tuple (PC_C, PC_1, PC_2, $\cdots$, PC_m) where $PC_C \in S_C, PC_1 \in S_1, \cdots, PC_m \in S_m$, indicating processes PC_C, PC_1, PC_2, $\cdots$, PC_m run simultaneously on core C

Table 1: Performance Model Validation

Benchmark		gzip	vpr	mcf	bzip2	twolf	art	equake	ammp	Avg.
MPA	E (%)	0.16	2.54	1.33	2.97	1.91	1.33	0.42	3.48	1.76
	>5% (%)	0	0	0	25	0	0	0	12.5	4.69
SPI	E (%)	0.58	5.58	2.15	2.06	4.54	5.51	1.89	3.80	3.38
	>5% (%)	0	50	0	0	37.5	50	12.5	25	21.9

and its partners core 1 through core m. For the set of process combinations that do not include process K, denoted as S_{ex}, the average power consumption, denoted by P_{ex}, is the sum of current power consumptions of core C and cores in PS_C. On the other hand, if we use S_{in} to represent the set of process combinations that include process K, for each item I in S_{in}, we use the performance model to predict the SPI and L2MPS for each process that belongs to I, which are then fed into the power model to calculate the corresponding power consumption for the process combination I. We use P_{in} to denote the average power consumptions for all combinations in S_{in}. Hence, the processor power consumption $P_{\text{processor}}$ can be written as

$$P_{\text{processor}} = (N-m)\cdot P_{\text{idle}} + \frac{P_{\text{ex}}\cdot|S_{\text{ex}}| + P_{\text{in}}\cdot|S_{\text{in}}|}{|S_{\text{ex}}| + |S_{\text{in}}|} + P_{\text{rest}}, \tag{11}$$

where P_{rest} is the current power consumption of cores that do not share cache with core C. Therefore, by profiling each process individually, we are able to estimate the processor power consumption for any process-to-core mapping, reducing the exponential time complexity for a simulation based approach to linear time complexity.

6. EXPERIMENTAL RESULTS

In this section, we first describe the experimental setup for model validation. We then present the validation results for the performance model, the power model, and the combined model.

6.1 Experimental Setup

We use PAPI 3.6.2 [2] to sample the HPCs. The sampling period is 30 ms. Our testsuite includes 8 SPEC CPU2000 benchmarks that compiled on the test system using gcc 4.1. This set contains both memory-intensive and CPU-intensive benchmarks. We record the program phase information for each benchmark during profiling. Experimental results indicate all but two benchmarks have only one significant phase, as defined by our parameters of interest. The longest phases in *art* and *mcf* were used (refer to Tam et al. [10] for details).

To determine power consumption, we use a Fluke i30 current clamp on one of the 12 V processor power supply lines, the output of which is sampled by an NI USB6210 data acquisition card. An on-chip voltage regulator converts this voltage to the actual processor operating voltage. We assume a fixed regulator efficiency of 90%. Therefore, $P = 0.9V\cdot I = 10.8\cdot I$, where P is the processor power and I is the measured current. The data acquisition card samples at a frequency of 10 kHz in our experiments.

6.2 Performance Model Validation

We first validate the performance model on an Intel Core2 Quad core Q6600 processor with two dies (and two cores per die) and 8 MB 16-way set-associative L2 cache in total, which runs Linux 2.6.28 (denoted as "4-core server"). As explained in Section 3.4, we first obtained the feature vectors of all 8 benchmarks using the stressmark. The performance model then takes the feature vector of each process for performance prediction. We measured all 36 possible pairwise combinations of 8 benchmarks: each benchmark is paired with every other benchmark (including another instance of itself) and assigned to two cache-sharing cores. The measured performance data are then compared to those predicted by our performance model.

Table 1 presents the average prediction error in MPA and SPI for each benchmark when it runs simultaneously with each of the 8 benchmarks. Row 2 shows the average absolute estimation error in MPA based on our model. Row 4 indicates the average relative estimation error in SPI. Row 3 and 5 present the percentage of test cases with an estimation error larger than 5% among all 8 test cases for each benchmark. The last column corresponds to the average result of all 8 benchmarks, i.e., 36 testcases. As indicated in Table 1, our technique has an average of 3.38% SPI estimation error across all 8 benchmarks with only 21.9% of the cases having an estimation error greater than 5%. The performance model is also validated on another Intel Core 2 Duo P6800 processor with two cores and 3 MB 12-way associative L2 cache using 55 combinations of 10 SPEC benchmarks. The average SPI estimation error is 1.57%.

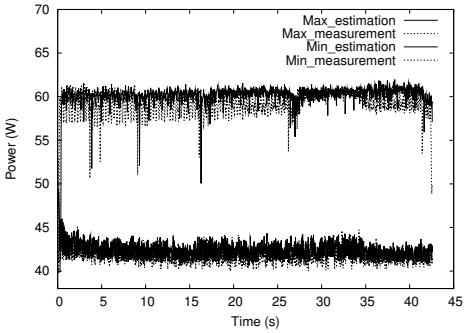

Figure 2: Power model validation on 4-core server.

Table 2: Power Model Validation on a 2-Core Workstation

Scenarios	Number of assignments	Avg./max. error for power samples (%)	Avg./max. error for avg. power (%)
1 proc./core	36	5.32 / 14.12	3.63 / 13.83
2 proc./core	24	6.65 / 8.84	2.47 / 4.05

6.3 Power Model Validation

We validated our power models on (1) a Pentium Dual Core E2220 processor with 1 MB L2 cache, which runs Linux 2.6.25 and (2) a 4-core server. For each machine, we first build the power model using 8 SPEC CPU2000 benchmarks and the customized micro-benchmark as explained in Section 4.1. We then validate the power model by assigning a combination of several SPEC CPU2000 benchmarks to some or all of the cores and compare the real power consumption with the power estimations using HPC values gathered during runtime. Note that we only analyze the duration in which all processes assigned are running concurrently.

Figure 2 illustrates the sample-based power model validation on the 4-core server for the assignments with the maximum and the minimum average power among all test cases.

Table 3: Power Model Validation on a 4-Core Server

Scenarios	Number of assignments	Avg./max. error for power samples (%)	Avg./max. error for avg. power (%)
1 proc./core	24	4.09 / 8.52	3.26 / 7.71
2 proc./core	3	5.51 / 6.25	4.47 / 5.95
4 proc. with unused cores	10	3.39 / 4.73	2.54 / 4.14

Table 4: Validating the Combined Model on a 4-Core Server

Scenarios	Number of assignments	Avg./max. error for avg. power (%)
1 proc./core	32	2.84 / 5.78
2 proc./core	10	1.92 / 6.29
4 proc., 1 core unused	16	2.68 / 5.48
4 proc., 2 core unused	16	2.53 / 5.99
4 proc., 3 core unused	9	0.49 / 1.95

The X axis is time and the Y axis is the power consumption. The solid lines represent power estimations, while the dotted lines represent measured values. They generally overlap, indicating good estimation accuracy. The average estimation errors are 2.46% and 2.51% for the maximum-power scenario and the minimum-power scenario, respectively.

Table 2 and Table 3 show the validation results for the power model on the 2-core workstation and 4-core server, respectively. Column 1 shows the testing scenario, e.g., "1 proc./core" refers to assignment schemes in which all cores are used with one SPEC program per core. Column 2 represents the number of different assignments evaluated given the testing scenario indicated in Column 1. Note that the processes in each assignment are chosen randomly in order to test the model on a wide range of scenarios. Column 3 presents the average and maximum error resulting from comparing the estimated processor power with the measured power for all power estimation samples. Column 4 presents the average and maximum error resulting from comparing the estimated average power with the measured average power.

On the 2-core workstation, we tested 36 different assignments with 1 process per core and 24 assignments with 2 processes per core. For a sample-based comparison, the average error for both scenarios are 5.32% and 6.65%, with maximum errors of 14.12% and 8.84%. For an average-power–based comparison, the average error for both scenarios are 3.63% and 2.47%, with maximum errors of 13.83% and 4.05%.

On the 4-core server, we tested 24 different assignments with 1 process per core, 3 assignments with 2 processes per core, and 10 assignments with 1 or 2 cores unused. For a sample-based comparison, the average error for the three scenarios are 4.09%, 5.51%, and 3.39%, with maximum errors of 8.52%, 6.25%, and 4.73%. For an average power comparison, the average errors for the three scenarios are 3.26%, 4.47%, and 2.54%, with maximum errors of 7.71%, 5.95%, and 4.14%. Therefore, we conclude the proposed power model is accurate and is sufficiently general to be used for different architectures, although the limited number of architectures considered is not sufficient to determine the were the limits on generality are located.

6.4 Combined Model Validation

We validated the combined performance and power model for average power estimation during assignment on the 4-core server. We first built the performance model and the power model as explained in Section 3 and Section 4. We then estimated the power consumption of an assignment following the algorithm in Figure 1. Note that only profiling information are used for estimation. The estimated average power is then compared to the measured average power to determine the accuracy of the combined model.

We tested 32 assignments with 1 process assigned to each core, 10 assignments with 2 processes assigned to each core, 16 assignments with 4 processes assigned to 3 cores, 16 assignments with 4 processes assigned to 2 cores, and 9 assignments with 4 processes assigned to a single core. The average errors for the 5 scenarios were 2.84%, 1.92%, 2.68%, 2.53%, and 0.49%, while the maximum errors were 5.78%, 6.29%, 5.48%, 5.99%, and 1.95%. We thus conclude that the combined model is effective in estimating the processor power consumption during assignment.

7. CONCLUSIONS

Accurately modeling the performance and power consumption in a multi-programmed multi-core environment is challenging but essential for optimizing process assignment and migration. This paper describes an on-line performance and power modeling framework that rapidly and accurately estimates the power consumption and performance implications of particular process-to-core mappings. This process requires no changes to existing operating system or hardware. The individual models and the combined model have been validated on multiple CMP machines with distinct architectures and nominal power consumptions. We conclude that the proposed framework is effective for performance and power estimation during both process assignment and execution.

8. REFERENCES

[1] Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

[2] PAPI 3.6.2. http://icl.cs.utk.edu/papi/.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. Int. Symp. Computer Architecture*, pages 83–94, June 2000.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 340–351, Feb. 2005.

[5] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pages 18–28, Dec. 2004.

[6] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 221–226, Aug. 2005.

[7] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. Int. Symp. Microarchitecture*, pages 78–88, Dec. 2006.

[8] K. Singh, M. Bhadhauria, and S. McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, pages 46–55, May 2008.

[9] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 117–128, Feb. 2002.

[10] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 121–132, Mar. 2009.

[11] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Proc. Int. Conf. Performance Analysis of Systems and Software*, Mar. 2010. To appear.