Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data

Canturk Isci and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{canturk,martonosi}@ee.princeton.edu

Abstract

With power dissipation becoming an increasingly vexing problem across many classes of computer systems, measuring power dissipation of real, running systems has become crucial for hardware and software system research and design. Live power measurements are imperative for studies requiring execution times too long for simulation, such as thermal analysis. Furthermore, as processors become more complex and include a host of aggressive dynamic power management techniques, per-component estimates of power dissipation have become both more challenging as well as more important.

In this paper we describe our technique for a coordinated measurement approach that combines real total power measurement with performance-counter-based, per-unit power estimation. The resulting tool offers live total power measurements for Intel Pentium 4 processors, and also provides power breakdowns for 22 of the major CPU subunits over minutes of SPEC2000 and desktop workload execution. As an example application, we use the generated component power breakdowns to identify program power phase behavior. Overall, this paper demonstrates a processor power measurement and estimation methodology and also gives experiences and empirical application results that can provide a basis for future power-aware research.

1. Introduction

Energy and power density concerns in modern processors have led to significant computer architecture research efforts in power-aware and temperature-aware computing. As with any applied, quantitative endeavors in architecture, it is crucial to be able to characterize existing systems as well as to evaluate tradeoffs in potential designs.

Unfortunately, cycle-level processor simulations are time-consuming, and are often vulnerable to concerns about accuracy. In certain cases,very long simulations can be required. This is particularly true for thermal studies, since it takes a long time for processors to reach equilibrium thermal operating points [24]. Furthermore, researchers often need the ability to measure live, running systems and to correlate measured results with overall system hardware and software behavior. Live measurements allow a complete view of operating system effects, I/O, and many other aspects of "real-world" behavior, often omitted from simulation.

While live measurements gain value from their completeness, it can often be difficult to "zoom in" and discern how different subcomponents contribute to the observed total. For this reason, many processors provide hardware performance counters that help give unit-by-unit views of processor events. While good for understanding processor *performance*, the translation from performance counters to power behavior is more indirect. Nonetheless, some prior research efforts have produced tools in which per-unit energy estimates are derived from performance counters [16, 17].

Prior counter-based energy tools have been geared towards previous-generation processors such as the Pentium Pro. Since these processors used little clock gating, their power consumption varied only minimally with workload. As a result, back-calculating unit-by-unit power divisions is fairly straightforward. In today's processors, however, power dissipation varies considerably— 50 Watts or more—on an application-by-application and cycle-by-cycle basis. As such, counter-based power estimation warrants further examination on aggressively-clock-gated superscalar processors like the Intel Pentium 4.

The primary contributions of this paper are as follows:

- We describe a detailed methodology for gathering live, per-unit power estimates based on hardware performance counters in complicated and aggressively-clock-gated microprocessors.
- We present live total power measurements for SPEC benchmarks as well as some common desktop applications.
- As an application of the component power estimates, we describe a power-oriented phase analysis using Bayesian similarity matrices.

The remainder of this paper is structured as follows. Section 2 gives an overview of our performance counter and

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE

power measurement methodology. Sections 3 and 4 then go into details about our mechanisms for live monitoring of performance counters and power. Following this, Section 5 develops a technique for attributing power to individual hardware units like caches, functional units, and so forth by monitoring appropriate performance counters. Section 6 gives results on total power and per-unit power measurements for collections of SPEC and desktop applications, and Section 7 gives an example use of component power estimates to track program power phases. Finally, Section 8 discusses related work and Section 9 gives conclusions and offers ideas for future work.

2. Overall Methodology

The fundamental approach underlying our power measurements is to use sampled multimeter data for overall total power measurements, and to use estimates based on performance counter readings to produce per-unit power breakdowns. Figure 1 shows our basic approach.

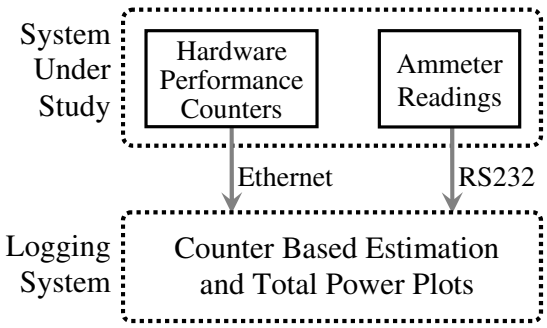


Figure 1. Overall power measurement and estimation system flow.

Live power measurements for the running system are obtained from a clamp ammeter measuring current on the appropriate power lines. The clamp ammeter plugs into a digital multimeter for data collection. A separate logging machine queries the multimeter for data samples and then generates runtime power plots and power logs for arbitrarily long timescales. Section 4 describes the power measurement setup in more detail and presents several power traces to demonstrate the power measurement scheme.

In parallel, per-unit power estimates are derived from appropriate weightings of Pentium 4 hardware performance counter readings. To access the P4 performance counters, there are a small number of pre-written counter libraries available [10, 25]. For efficiency and ease-of-use we have written our own Linux loadable kernel module (LKM) to access the counters. Our LKM-based implementation offers a mechanism with sufficient flexibility and portability, while incurring almost zero power and performance overhead so that we can continuously collect counter information at runtime and generate runtime power statistics.

Section 5 describes our power estimation technique based on the performance counter readings obtained from a live program run. From a Pentium 4 die photo, we break the processor into sub-units such as L1 cache, branch prediction hardware, and others. For each component, we develop a power estimation model based on combinations of events available to P4 hardware counters as well as heuristics that translate event counts into approximate access rates for each component. Some components, such as caches, are relatively straightforward since there are hardware event counters that map directly to their accesses. Other components, such as bus logic, have less straightforward translations, as discussed in Section 5. As the last step, we use the real power measurements obtained from the ammeter in conjunction with logging system's counter-based power estimates to synchronize and provide a comparison between the measured and estimated total power measures.

The machine used in our power measurement and estimation experiments is a 1.4GHz Pentium 4 processor, 0.18μ Willamette core. The CPU operating voltage is 1.7V and published typical and maximum powers are 51.8W and 71W, respectively [13]. The NetBurst microarchitecture of P4 is based on a 20-stage misprediction pipeline with a trace cache to remove instruction decoding from the main pipeline. In addition to a front-end BPU a second smaller BPU is used to predict branches for uops within traces. It has two double-pumped ALUs for simple integer operations. L1 cache is accessed in 2 cycles for integer loads, while the L2 cache is accessed in 7 cycles.

The processor implements extremely aggressive power management, clock gating, and thermal monitoring. Almost every processor unit is involved in power reduction and almost every functional block contains clock gating logic, summing up to 350 unique clock gating conditions. This aggressive clock gating provides up to approximately 20W power savings on typical applications [4].

The following sections present our runtime power modeling methodology in progressive manner. We first describe event monitoring with performance counters, then Section 4 describes the real power measurement setup and finally Section 5 discusses the power estimators we have developed.

3. Using Pentium 4 Performance Counters

Most of today's processors include some dedicated hardware performance counters for debugging and measurement. In general, performance counter hardware includes event signals generated by CPU functional units, event detectors detecting these signals and triggering the counters, and configured counters incrementing according to the triggers.

Intel introduced performance counting into IA32 architecture with Pentium processors and has gone through two more counter generations since then. The Pentium 4 performance counting hardware includes 18 hardware counters that can count 18 different events simultaneously and

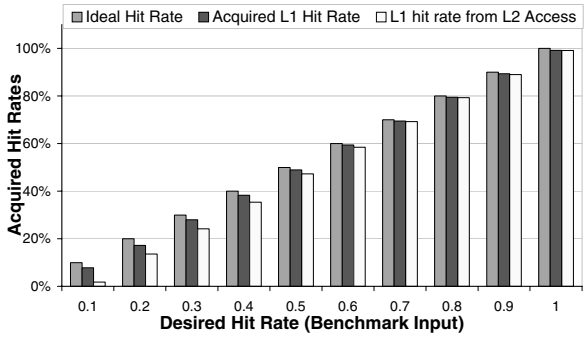


Figure 2. L1 hit rate validation.

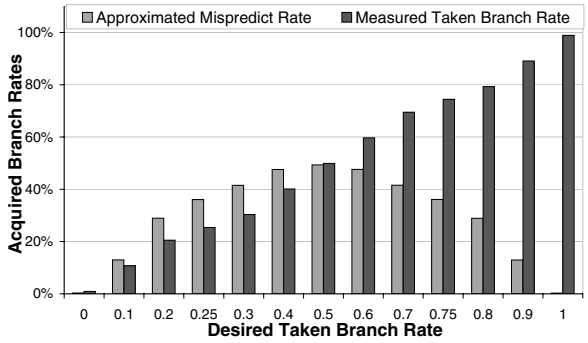


Figure 3. Branch rate validation.

in parallel with pipeline execution. 18 counter configuration control registers (CCCRs), each associated with a unique counter, configure the counters for specific counting schemes such as event filtering and interrupt generation. 45 event selection control registers (ESCRs) specify the P4 events to be counted and some additional model specific registers (MSRs) for special mechanisms like replay tagging. In addition to the 18 event counters, there exists a special time stamp counter (TSC) that increments with processor clock cycle [11, 26]. Intel P4 performance monitoring events comprise 59 event classes that enable counting several hundred specific events as described in Appendix A of [11]. The event metrics are broken into categories such as general, branching, trace cache and front end, memory, and others [12]. A more comprehensive description of event detection and counting can be found in [26].

In order to use the performance counters, we implement two LKMs. The first LKM, *CPUinfo*, is simply used to read information about the processor chip installed in the system being measured. This helps the tool identify architecture specifications and discern the availability of performance monitoring features. The second LKM, *PerformanceReader*, implements six system calls to specify the events to be monitored, and to read and manipulate counters. The system calls are: (i) **select events:** Updates the ESCR and CCCR fields as specified by the user to define the events, masks, and counting schemes, (ii) **reset event counter:** Resets specified counters, (iii) **start event counter:** Enables specified counter's control register to begin counting, (iv) **stop event counter:** Disables specified counter's control register to end counting, (v) **get event counts:** Copies the current counter values and time stamp to user space, and (vi) **set replay MSRs:** updates special MSRs required for "replay tagging".

The performance reader is very lightweight and low-overhead. The start, stop, and reset event counter system calls are only three assembly instructions. The system call for getting event counts at the end of a measurement is longer as it requires copying data elsewhere in memory, but need not be executed during the core of an application. Because the performance reader has a simple and lightweight interface, we can completely control and update counters easily from within any application.

To validate the performance reader, we wrote microbenchmarks targeting specific processor units. The first benchmark, shown in Figure 2, generates a desired L1 cache hit rate by executing 1 billion iterations of traversal through a large linked list of pointers in a pseudo-random sequence, with the sequence length guided by the user-specified desired hit rate. We use two metrics to evaluate the L1 hit rates. The first metric uses an event counter directly counting load instructions that are tagged due to a load miss replay. The second, less direct, metric uses L2 accesses as a proxy for L1 misses as long as data is expected to reside in L2 cache and other application/system accesses to L2 are minimal. The figure shows that both methods of measuring L1 cache performance track the target quite closely, particularly for hit rates of 50% or more. Both counter-measured hit rates are slightly below the program's target; this is due to initialization effects, conflicts and sharing of cache among processes. In the main benchmark loop of the cache experiment, there are 8 IA32 instructions, and so we expect 8 billion retired instructions from the full program. The actual value read from the counters is $8.011 \cdot 10^9$, where most of the additional instructions are due to OS scheduling. Thus, the performance reader operates accurately and with trivial overhead, as long as sampling intervals are kept on the order of milliseconds.

The second benchmark, illustrated in Figure 3, generates a desired rate of taken branches by comparing a large random data set to a threshold set to generate the desired branching rate. Moreover, the randomness of the data enables us to approximately specify the expected mispredict rate as: $MispredictRate = 0.5 - |TakenBranchRate - 0.5|$. As the figure shows, the branch microbenchmark produces the desired amount of taken branches effectively. Additionally, the mispredict rates generated are closely related to our expectation, usually shooting around 10% higher in 20-40% expected misprediction range.

4. Real Power Measurements

In Figure 4 we show the details of our power measurement setup, where CPU power is measured with the clamp ammeter. While some prior work has used series or shunt resistors to measure power dissipation [21, 28], we chose the clamp ammeter approach because it avoids the need to cut or add any wires on the system being measured.

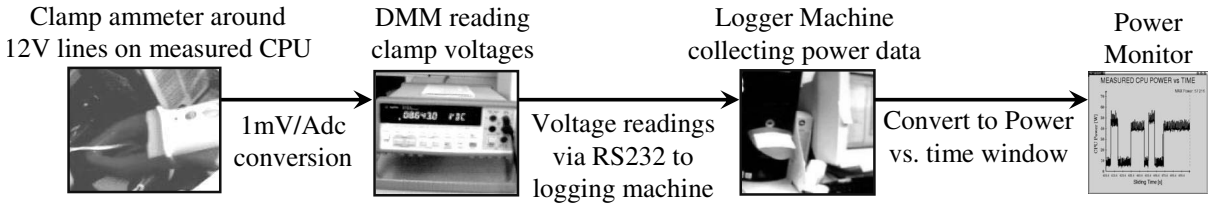


Figure 4. Processor power measurement setup.

The main power lines for the CPU operate at 12V, and then are fed to a voltage regulator module, which converts this voltage to the actual processor operating voltage and provides tight control on voltage variations [30]. Therefore, we use our current probe to measure the total current through the 12V lines. To verify our setup, we measured the current through each of the 17 power lines in our system, while running a microbenchmark that creates high fluctuations in processor power. In accordance with our choice of power lines, three 12V lines yielded current variations that followed the processor activity tracked by the performance reader, while other lines presented an uncorrelated power behavior, usually with insignificant power variation.

We use a Fluke i410 current probe connected to an Agilent 34401 digital multimeter (DMM). The DMM sends the voltage readings to a second logging machine, a 2.2GHz Pentium 4 Processor, via the serial port. The logger machine gets voltage data from serial port and converts these values into processor power dissipation with the power relation: $P = V \cdot I = 12 \cdot (VoltageSample[V]) \cdot 1000$. It then displays the measured runtime power in a *power monitor* with a sliding time window, while also logging time vs. voltage information.

In our experiments, we sample 1000 readings per second with $4\frac{1}{2}$ digit resolution, which corresponds to 0.12W power resolution. The DMM can generate around 55 ASCII readings per second to RS232, so we collect the data in the logger machine at 20ms intervals. The logging machine then computes a moving average within an even longer second sampling period that is used to update the on-screen power monitor and the data log. (We use this second sampling period so that we can likewise use coarse sampling periods to read performance counters in Section 5.)

We close this section with a selection of total power observations from the SPEC benchmarks. In Figure 5, we demonstrate the power traces for Spec2000 benchmarks gcc and vpr, compiled with gcc-2.96 with -O3 -fomit-frame-pointer compiler flags, for *full* benchmark runs over several minutes. Despite their similar average powers (see Figure 14 (a), non-idle average measured power), the two benchmarks show a very different power behavior during their runtimes. The vpr benchmark maintains a very stable power, while gcc produces significant power fluctuations. The applications also clearly demonstrate several phases of execution during their lifetimes. Given the long timescales over which these data were collected, it is clear that live power measurements are crucial to future power-aware research, since simulation times to gather such data would be prohibitive.

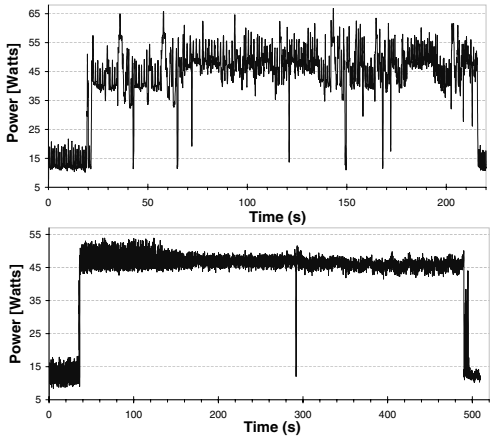


Figure 5. SPEC gcc (upper) and vpr (lower) power traces.

5. Modeling Power for Processor Sub-Units

While total power measurements for long-running programs are already useful, we also wish to be able to estimate how power subdivides among different hardware units. Prior work has developed counter-based or profile-based estimates for much simpler processors [3, 16, 19, 28]. In our approach, we estimate physical component powers using counter-based measures, and also generate reasonable total power estimates.

Our modeling technique is distinct from prior work in the following ways. We estimate power for a much more complicated modern processor, with extremely aggressive clock gating and high power variability. Second, we consider strictly physical components directly from the die layout, as opposed to "proxy" categories aggregated for convenience but not present as a single hardware unit on the die. Finally, we estimate power for all levels of processor utilization for arbitrarily long periods of time, rather than restricting our technique only to power variations at high processor utilization. The latter two are particularly important for thermal studies. In this section, we first walk through our methodology and then demonstrate the experimental setup.

5.1. Defining Components for Power Breakdowns

The processor components for which the power breakdowns are generated might be chosen in different ways, with varying granularity and interpretations. For example, one can consider the four processor subsystems described in [8]: memory, in-order front end, out of order engine, and

execution. Or instead, one might use a more conceptual interpretation similar to [16] in which categories such as issue, execution, memory, fetch and reorder-related are considered. The first option lacks the fine granularity we desire, while the second doesn't provide a direct mapping to a physical layout.

In our approach, we choose a reasonably fine granularity, and—most importantly—our categories are strictly colocated physical components identifiable on a die photo. This decision is based on the ultimate endgoal of our project which is to support thermal modeling and processor temperature distributions, both of which rely on actual processor physical parameters. Consequently, based on an annotated P4 die photo we define 22 physical components: Bus control, L1 cache, L2 cache, L1 branch prediction unit (BPU), L2 BPU, instruction TLB & fetch, memory order buffer, memory control, data TLB, integer execution, floating point execution, integer register file, floating point register file, instruction decoder, trace cache, microcode ROM, allocation, rename, instruction queue1, instruction queue2, schedule, and retirement logic.

5.2. P4 Counter Events to Drive Estimation

For each of the 22 components, we need a performance counter event or a combination of events that can approximate the access count of that component. The finalized set of heuristics that define these access counts involve 24 event metrics composed in various ways for the 22 defined processor components. While the full set of heuristics is too large to present here, Table 1 gives a sample of processor components and the corresponding performance counter metrics we devised. The full complement is available in [15].

For example, bus control access rates can be obtained by configuring IOQ Allocation to count all bus transactions (all reads, writes and prefetches) that are allocated in the IO Queue (between the L2 cache and bus sequence queue) by all agents (all processors and DMAs). FSB data activity is configured to count all DataReaDY and DataBuSY events on the front side bus, when processor or other agents drive/read/reserve the bus. The bus ratio (3.5 in our implementation) is the ratio of processor clock (1400MHz) to bus clock (400MHz), and converts the counts in reference to processor clock cycles.

Trace cache activity can be discerned by configuring the "Uop queue writes" metric to count all speculative uops written to the small in-order uop queue in front of the out-of-order engine. These come from either trace cache build mode, trace cache deliver mode or microcode ROM.

As a final example, there is no direct counter event for the total number of integer instructions executed. Instead, we total up the counters for the eight types of FP instructions, giving us an estimate of total FP operations issued. We then subtract this from the total written speculative uops to get an integer estimated total. Integer operations that are not load/store or branch are scaled by 2 as they use the double pumped ALU. On the other hand, load/stores use address generation units and branch processing is done in the complex ALU, along with shifts, flag logic and multiplies. The counters do not let us differentiate multiply and shifts and therefore they are also scaled by 2. Also, some x87 and SIMD instructions are decoded into multiple uops, which may cause undercounting.

Ultimately, we use 15 counters with 4 rotations. The P4 events and counter assignments minimize the counter switches required to measure all the metrics needed. At least four rotations are unavoidable. This is because floating point metrics involve 8 different events, of which only two at a time can be counted due to the limitations of P4 counters in ESCR assignments.

5.3. Counter-based Component Power Estimation

We use the component access rates—either given directly by a performance counter or approximated indirectly by one or more performance counters— to weight component power numbers. In particular, we use the access rates as weighting factors to multiply against each component's maximum power value along with a scaling strategy that is based on microarchitectural and structural properties. In general, all the component power estimations are based on Equation 1, where maximum power and conditional clock power are estimated empirically during implementation. The C_i in the equation are the hardware components, 1 through 22.

$$\begin{aligned} Power(C_i) &= AccessRate(C_i) \cdot \\ &\quad ArchitecturalScaling(C_i) \cdot \\ &\quad MaxPower(C_i) + \\ &\quad NonGatedClockPower(C_i) \end{aligned} \quad (1)$$

Of the 22 components, six issue logic units (trace cache, allocation, rename, schedule and both instruction queues) require a special piecewise linear approach. This is because of nonlinear behavior, in which an initial increase from idle to a relatively low access rate causes a large increase in processor power, while further increases in access rates produce much smaller power variations. Therefore, for the issue logic components, we apply a conditional clock power factor in addition to linear scaling above a low threshold value.

As an example of our overall technique, consider the trace cache. It delivers three uops/cycle when a trace is executed and builds one uop/cycle when instructions are decoded into a trace. Therefore, the access rate approximation in deliver mode is scaled by 1/3, while access rate from instruction decoder is scaled with 1. These rates are then used as the weighting factor for estimated maximum trace cache power.

We construct the total power, shown in Equation 2, as the sum of 22 component powers calculated as above, along with a fixed idle power of 8W from the total power measurements described in Section 4. Hence, this fixed 8W base includes some portion of globally non-gated clock power, whereas the conditionally-gated portion of clock power is

Bus Control	$\frac{IOQ\ Allocation}{\Delta Cycles_1} + \frac{Bus\ Ratio \cdot FSB\ Data\ Activity}{\Delta Cycles_2}$
Front End BPU	$\frac{8 \cdot ITLB\ Reference}{\Delta Cycles_1} + \frac{Branch\ Retired}{\Delta Cycles_2}$
Secondary BPU	$\frac{Branch\ Retired}{\Delta Cycles_2}$
L1 Cache	$\frac{Ld\ Port\ Replay + St\ Port\ Replay}{\Delta Cycles_1} + \frac{Front\ End\ Event}{\Delta Cycles_2}$
MOB	$\frac{MOB\ Load\ Replay}{\Delta Cycles_2}$
Trace Cache	$\frac{Uop\ Queue\ Writes}{\Delta Cycles_1}$
Integer Execution	$2 \cdot (\frac{Uop\ Queue\ Writes}{\Delta Cycles_1} - FP\ Exe.\ Access\ Rate) - L1\ Cache\ Access\ Rate - \frac{Branch\ Retired}{\Delta Cycles_2}$
L2 Cache	$\frac{BSQ\ Cache\ Ref}{\Delta Cycles_1}$
DTLB	$L1\ Cache\ Access\ Rate + MOB\ Access\ Rate$
ITLB	$\frac{ITLB\ Ref}{\Delta Cycles_1} + \frac{BPU\ Fetch\ Req}{\Delta Cycles_2}$

Table 1. Examples of processor components and access rate heuristics, which are used as corresponding power weightings for the components.

distributed into component power estimations.

$$Total\ Power = \sum_{i=1}^{22} Power(C_i) + Idle\ Power \qquad (2)$$

For initial estimates of each component's "maxpower" value, $MaxPower(C_i)$ in Equation 1, we used physical areas on the die. We scaled the documented maximum processor power by the component area ratios. In many cases, these areas serve as good proportional estimates. To further tune these maximum power estimates, we developed a small set of training benchmarks that exercise the CPU in particular ways. By measuring total power with a multimeter, we could compare true total power over time to the total power estimated by summing our component estimates. We thus observed how diferent settings of "maxpower" weight factors affect the accuracy of our power estimations for the training benchmarks. After several experiments with the training benchmarks, we arrived at a final set of maxpower and non-gated clock power values for each of the components. These are hard-coded as the P4 specific weighting factors in the final implementation of our power estimation setup.

Figure 6 shows a screenshot of the resulting power estimations for the training benchmarks, together with the measurement data in the combined total power monitor. The dark colored estimated power is plotted synchronously with the lighter colored measured runtime power. The leftmost benchmark, *fast*, is very simple code with two integer and one floating-point operation inside a computation-dominated loop. The next two benchmarks are the *branch microbenchmark* and *cache microbenchmark* described in section 3. Finally, the last application in the timeline is called *hi-lo*, an iterative stressmark designed to repeatedly change power dissipation from very low to very high, which produces a roughly 30W power swing. One can see that the final maxpower settings lead to very good total power estimates over a wide range of power levels and application behaviors.

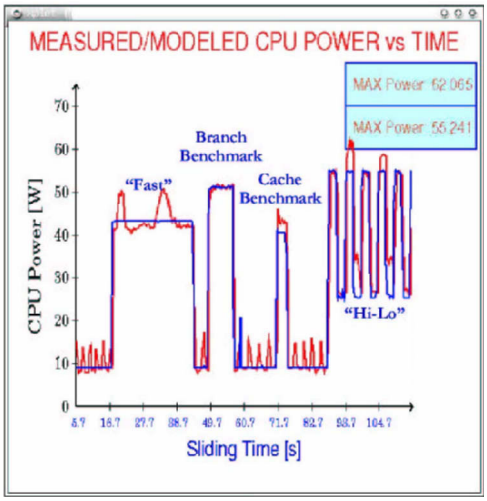


Figure 6. Total power after tuning of maximum component powers and idle power assumptions.

5.4. Final Implementation

To summarize, in our final implementation, we use our performance reader to provide the system with the required counter information and the logger machine collects all the counter and measurement information to generate the complete runtime component power modeling and total power verification system. We verify component based estimates against total power measurements using the setup in Figure 4.

Measured processor current is again sent by the DMM to the logger machine via RS232 and the logger machine converts the current information to power as before. On the measured machine, *PowerServer* collects counter information every 100 ms, for the P4 events chosen to approximate component access rates; it also applies counter rotations and timestamping. Every 400ms, it sends collected information to the logging machine over Ethernet. While this perturbs system behavior slightly, it is done as infrequently as possible to minimize the disturbance. On the logger machine,

PowerClient collects measured ammeter data from the serial port, and raw counter information from Ethernet. Combining the two, it applies the access rate and power model heuristics, and generates component power estimates for the defined components. After the processing of collected data, PowerClient generates the runtime component power breakdown monitor as well as runtime total power plots for both measured and counter estimated power after synchronizing the modeled and measured powers, over a 100 second time window, with an average update rate of 440ms.

5.5. Microbenchmark Results

To begin, we show generated power breakdowns for branch and cache microbenchmarks, that were introduced in Section 3. As one cannot gain physical access to per-component power to measure, and since per-component power values are not published, we use the close match of measured (ammeter) total power to estimated (counter-based) total power as a gauge of our model's accuracy.

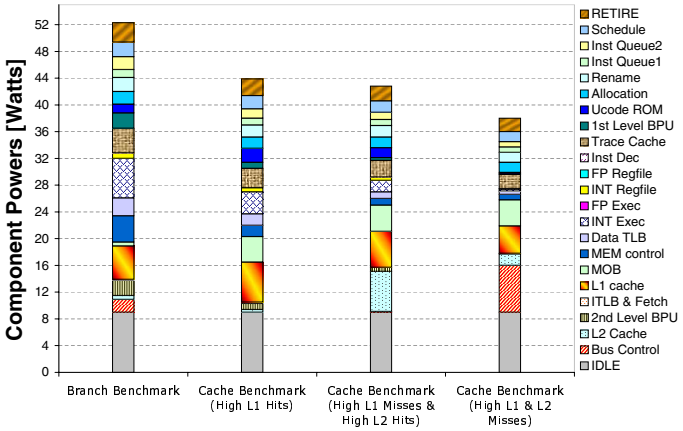


Figure 7. Power breakdowns for branch and cache benchmarks.

The leftmost bar of Figure 7 shows the estimated power breakdowns for our branch exercise microbenchmark. This is a very small program that is expected to reside mostly in trace cache and that is mostly L1 bound. This microbenchmark is a high uops per cycle (UPC), high-power integer program. The breakdowns show high issue, execution and branch prediction logic power, while as expected L2 cache and bus for main memory dissipate lower power.

Second bar of Figure 7 shows breakdowns for cache exercise microbenchmark with an almost perfect L1 hit rate. Once again, the component breakdowns track our intuition well. The breakdowns show high L1 power consumption and relatively high issue and execution power as we do not stall due to L1 miss and memory ordering/replay issues. Both L2 and bus power are relatively low.

In the third bar of Figure 7, we configure the cache microbenchmark to generate high L1 misses, while hitting almost perfectly in L2. The power distribution of L2 cache is seen to increase significantly, while execution and issue cores start to slow down due to replay stalls. Moreover memory order buffer shows slight increase due to increasing memory load and store port replay issues.

Finally, in the rightmost bar of Figure 7 we also generate high L2 misses and therefore bus power climbs up, while execution core slows down even further due to higher main memory penalties. Although total L2 accesses actually increase, due to significantly longer program duration, access rates related to L2 drop and aggregate L2 power decreases.

Overall, this sequence of microbenchmarks, while simple, builds confidence that the counter-based power estimates are showing reasonable accuracy. In the sections that follow, we present more large-scale, long-running experiments on SPEC and desktop applications.

6. Power Model Results

In the preceding section we showed some initial per-component power results for our microbenchmarks. Here, we provide power breakdowns and total power estimates for the full runtimes of selected SPEC benchmarks, as well as some practical desktop applications. Additional measurement and estimation data is available in [15]. The SPEC2000 benchmarks shown in this paper are compiled using gcc-2.96 and with compiler flags of "-O3 -fomit-frame-pointer". We use the reference inputs with a single iteration of run. In order to demonstrate our ability to model power closely even at low CPU utilizations, we also experimented with practical desktop tools: AbiWord for text editing, Mozilla for web browsing and Gnumeric for numerical analysis. All these benchmarks share the common property of producing low CPU utilization with only intermittent power bursts.

6.1. SPEC Results

In this section, we show SPECint programs vpr and twolf, and equake from SPECfp. Figures 8–13 show total power estimates and component power breakdowns for vpr, twolf and equake. Similar data for gcc are included in Figure 16. For reference inputs, the vpr benchmark actually consists of two separate program runs. The first run uses architecture and netlist descriptions to generate a placement file, while the second run uses the newly-generated placement file to generate a routing descriptor file [27]. Although the total average power for the two runs is quite similar, Figure 8 shows a noticeable phase change at around 300s when the second run begins. Figure 9 demonstrates even more clearly how distinct the power behavior in the second phase is. Although the first run, the placement algorithm, dissipates very stable power, the second phase's routing algorithm has a much more variable and periodic power behavior. As [18] discusses, the initial placement phase produces higher miss rates than the routing part. This is because routing benefits from the fact that placement brings much of the dataset into memory. The per-component power

breakdowns corroborate this: there is higher L1 power in first half due to memory ordering issues and increased L2 power in second phase. Although it is an integer benchmark, our breakdown also shows that vpr consumes significant amount of FP unit power. This is due to the SIMD instructions it employs which use the FP unit. (The counter x87_SIMD_moves_uops indicates a upc of 0.08 in placement and 0.22 in routing.)

Twolf is a transistor layout generation program that performs standard cell placement and connection. It performs several loop computations, traversing the memory and potentially producing cache misses. The high memory utilization of twolf is observed in the power breakdowns of Figure 11. Moreover, although twolf exhibits almost constant total power in Figure 10, individual component powers are not constant; there are slight increases in L1 cache and microcode ROM powers and decreases in L2 cache power over the runtime.

As an example of floating point benchmarks, we show the equake benchmark in Figures 12 and 13. Equake models ground motion by numerical wave propagation equation solutions [2]. The algorithm consists of mesh generation and partitioning for the initialization, and mesh computation phases. In Figure 12, we can already clearly identify the initialization phase and computation phase. Figure 13 demonstrates the high microcode ROM power as the initialization phase uses complex IA32 instructions extensively. The mesh computation phase, then exhibits the floating point intensive computations.

In addition to vpr, twolf and equake, we have generated similar power traces for several other Spec2000 benchmarks. Gcc is included in section 7 and counter based power estimations are seen to track the highly variant gcc power trace very favorably. Power traces for the rest of the investigated benchmarks are available in [15]. Figures 14 (a) and (b), however, present statistical measures that further confirm accuracy of our modeling framework, for a larger set of SPEC2000 benchmarks.

In Figure 14, we show the average powers computed from real power measurements and counter estimated total powers, for both the whole runtime of the benchmarks also including the idle periods and for the actual execution phases, excluding idle periods. Hence, the idle-inclusive measures cannot be considered as standard results, as the idle periods vary in each experiment - i.e. equake has a long idle period logged at the end of experiment, thus producing a very high standard deviation due to lowered full-runtime average power, around which the deviation is computed. They are of value, however, for comparing counter-based totals to measured totals, because one of our aims is to be able to characterize low utilization powers as well, with reasonable accuracy. For the estimated average powers, the average difference between estimated and measured powers is around 3 Watts, with the worst case being equake (Figure 12), with a 5.8W difference. For the standard deviation, the average difference between estimated and measured powers is around 2 Watts, with the worst case being vortex, with a standard deviation difference of 3.5W.

6.2. Desktop Applications

In addition to SPEC, we investigated three Linux desktop applications as well. These help demonstrate our power model's ability to estimate power behavior of practical desktop programs; because of their interactive nature, they typically present periods of low power punctuated by intermittent bursts of higher power. The three applications, shown in Figure 15, are AbiWord for text editing, Mozilla for web browsing and Gnumeric for numerical analysis.

In the web browsing experiment in Figure 15(a), the power traces represent opening the browser, connecting to a web page, downloading a streaming video and closing the browser. In the text editing experiment in Figure 15(b), the power traces represent opening the editor, writing a short text, saving the file and closing the editor. In the Gnumeric example in Figure 15(c), the power traces represent opening the program, importing a large text of 370K, performing several statistics on the data, saving the file and closing the program. The power traces reveal the bursty nature of the total power timeline for these benchmarks; this is particularly true at the moments of saving data to memory and computations. Overall, the long idle periods mean that the benchmarks have low average power dissipation. The power traces for the desktop applications also reveal that our counter based power model follows even very low power trends with reasonable accuracy. Together with the SPEC results, this demonstrates that our counter-based power estimates can perform reasonably accurate estimations independent of the range of power variations produced by different applications, without any realistic bounds on the observed timescale. To our knowledge, we are the first to produce live power measurements of this type for a processor as complex as the P4.

7. Power Phase Behavior Analysis

As a possible application of our technique, we demonstrate here how we can use component-based power breakdowns to identify *power* phases of programs. Several prior papers have proposed methods for detecting or exploiting program phases[1, 5, 9, 22, 23, 29]. Our example here is distinct because we focus on *power* phases rather than performance phases. A more detailed description of our power phase research can be found in [14].

We use the similarity matrix approach of [23] to deduce power phase behavior over the program runtime. We consider generated component breakdowns as coordinate vectors in the possible power space. Proceeding similarly to prior work on basic block vectors, we consider the manhattan distance between pairs of vectors as the "power behavior dissimilarity" between the two vectors. We do not normalize the power breakdown vectors, as our measure is already based on power and scaled power values should not be considered as similar power behavior. Consequently, we construct the power similarity matrix from manhattan distances of all the combination pairs of component power vec-

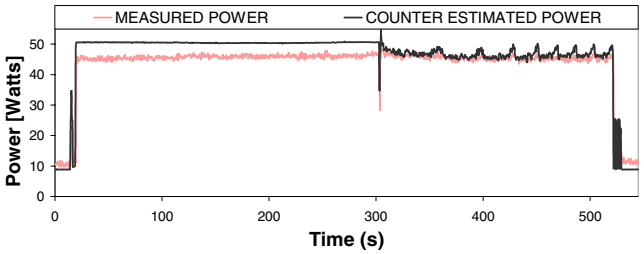


Figure 8. SPECint vpr total measured and modeled runtime power.

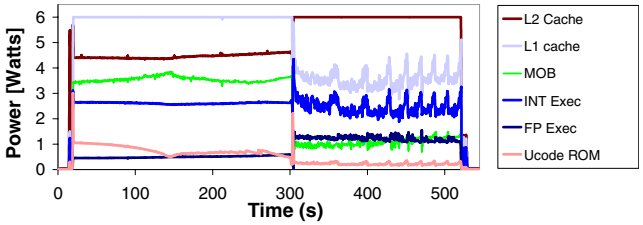


Figure 9. Estimated power breakdowns for vpr.

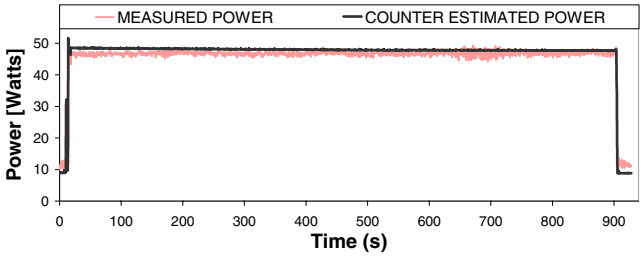


Figure 10. SPECint twolf total measured and modeled runtime power.

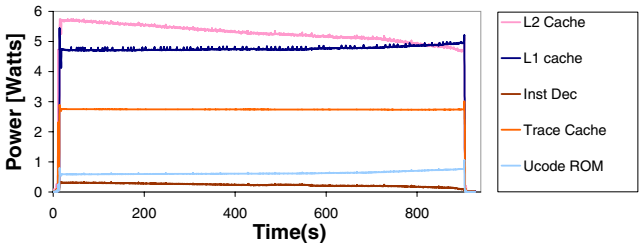


Figure 11. Estimated power breakdowns for twolf.

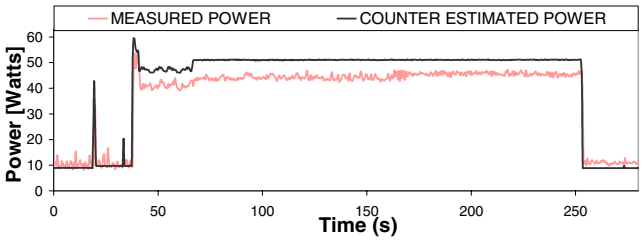


Figure 12. SPECfp equake total measured and modeled runtime power.

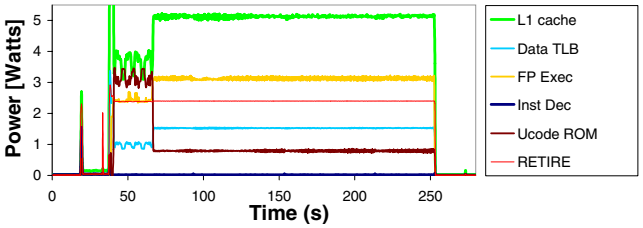


Figure 13. Estimated power breakdowns for equake.

tors. A single matrix entry is computed as shown in equation 3, where $Power_{r,c}$ represent the sample power vectors and C_i represent the individual components. The diagonal of the matrix represents the time axis and the entries to the right of the diagonal denote similarity between the current time sample and future samples. Entries above the current sample show similarity with respect to previous samples.

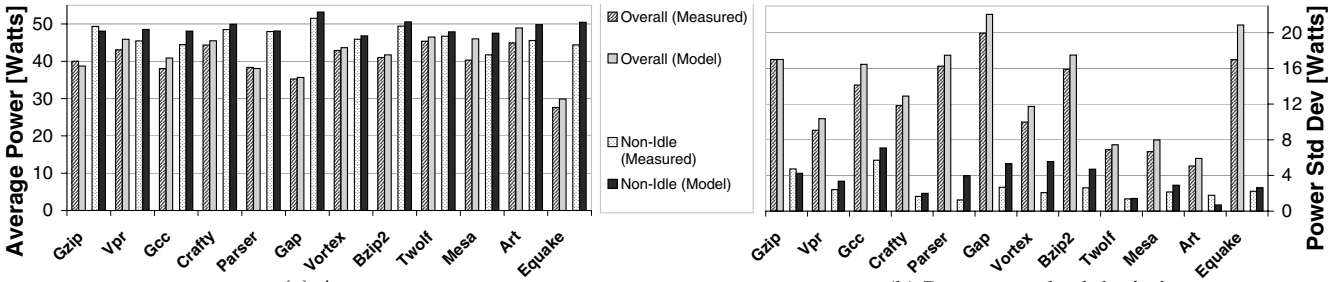
$$Similarity\ Matrix(r,c) = \sum_{i=1}^{22} |Power_r(C_i) - Power_c(C_i)| \quad (3)$$

In Figure 16, we present the acquired power phase similarity matrix for SPEC2000 benchmark gcc. In the uppermost matrix plot, the top left corner represents the start of the timeline and the lower right corner represents the end of the timeline. Darker regions represent higher similarity between the corresponding component power vectors. In the lower graphs of the figure, we also show the total power and component power breakdown estimations with the same timescales as the similarity matrix.

Even with gcc, which has very unstable power behavior, several similarities are highlighted by the similarity matrix. For example, consider the almost identical power behavior around the 30s, 50s and 180s points in the timeline. Moreover, by using component power breakdowns as power "signatures", the similarity matrix helps differentiate power behavior even in cases where the total power is measured to be quite similar. For example, although measured power is similar for gcc's regions around 88s, 110s, 140s, 210s and 230s, the similarity matrix reveals that the 88s, 210s and 230s regions are much more similar than the other two regions, which are also shown to be mutually dissimilar. On the other hand, other applications we have studied, such as gzip, show more regular patterns with several similar phases. By providing a foundation for power phase analysis, counter-based component power estimates are useful in power and thermal aware research. As our technique works at runtime, it is quite efficient and can be used to home in accurately on repetitive program phases even when little application information is available.

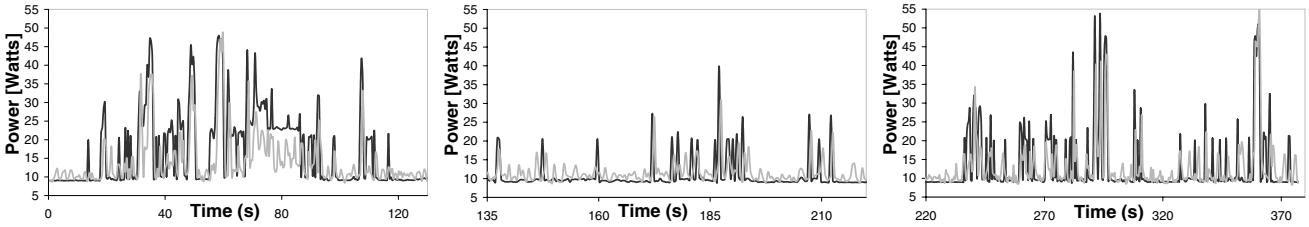
8. Related Work

While there has been significant work on processor power estimations, much has been based purely on simulations. Our approach, in contrast, uses live performance counter measurements as the foundation for an estimation technique.



(a) Average power (b) Power standard deviation

Figure 14. Average (left) and standard deviation (right) of measured and counter estimated power for SPEC2000 benchmarks. For each benchmark, the first set of power values represents averaging and standard deviation over the whole runtime of the program. The second set represents averaging and standard deviation only over non-idle periods.



(a) Mozilla (b) AbiWord (c) Gnumeric

Figure 15. Total measured (light) and counter estimated (dark) runtime power for 3 desktop applications.

One category of related work is research involving live measurements of total power. While these are numerous, we touch on a few key examples here. In early work, Tiwari et al. developed software power models for an Intel 486DX2 processor and DRAM and verified total power by measuring the current drawn while running programs [28]. They used the information to generate instruction energy cost tables and identify inter-instruction effects. Russell et al. likewise did software power modeling for i960 embedded processors, and validated using current measurements [20]. Flinn et al. developed the PowerScope tool, which maps consumed energy to program structure at procedural level [6]. This OS-oriented research uses a DMM to do live power measurements, and then uses energy analyzer software to attribute power to different processes or procedures.

More recently, Lee et al. used energy measurements based on charge transfer to derive instruction energy consumption models for a RISC ARM7TDMI processor [19]. They use linear regression to fit the model equations to measured energy at each clock cycle. These techniques are aimed at very simple processors with almost no clock gating, however, and therefore need to track and model only minimal cycle-by-cycle power variation. As a first example of Pentium 4 power measurement studies, Seng and Tullsen have investigated the effect of compiler optimizations on average program power, by measuring the processor power for benchmarks compiled with different optimization levels [21]. They use two series resistors in Vcc traces to measure the processor current. However, they do not present component power breakdowns or power-oriented phase analysis.

Next, we present prior work on performance counters and power metrics. Bellosa uses performance counters, to identify correlations between certain processor events, such as retired floating point operations, and energy consumption for an Intel Pentium II processor [3]. He proposes this counter-based energy accounting scheme as a feedback mechanism for OS directed power management such as thread time extension and clock throttling. Likewise, the Castle tool, developed by Joseph et al. [16], uses performance counters to model component powers for a Pentium Pro processor. It provides comparisons between estimated total processor power and total power measured using a series resistor in processor power lines. Our work makes significant extensions in both infrastructure and approach in order to apply counter-based techniques to a processor as complex as the P4. Furthermore, to our knowledge, neither Bellosa nor Joseph used their measurements to do phase analysis. Kadayif et al. use the Perfmon counter library to access performance counters of the UltraSPARC processor [17]. They collect memory related event information and estimate memory system energy consumption based on analytical memory energy model; they did not consider the rest of the processor pipeline. And finally, Haid et al. [7], propose a coprocessor for runtime energy estimation for system-on-a-chip designs. Essentially, the goal of that work

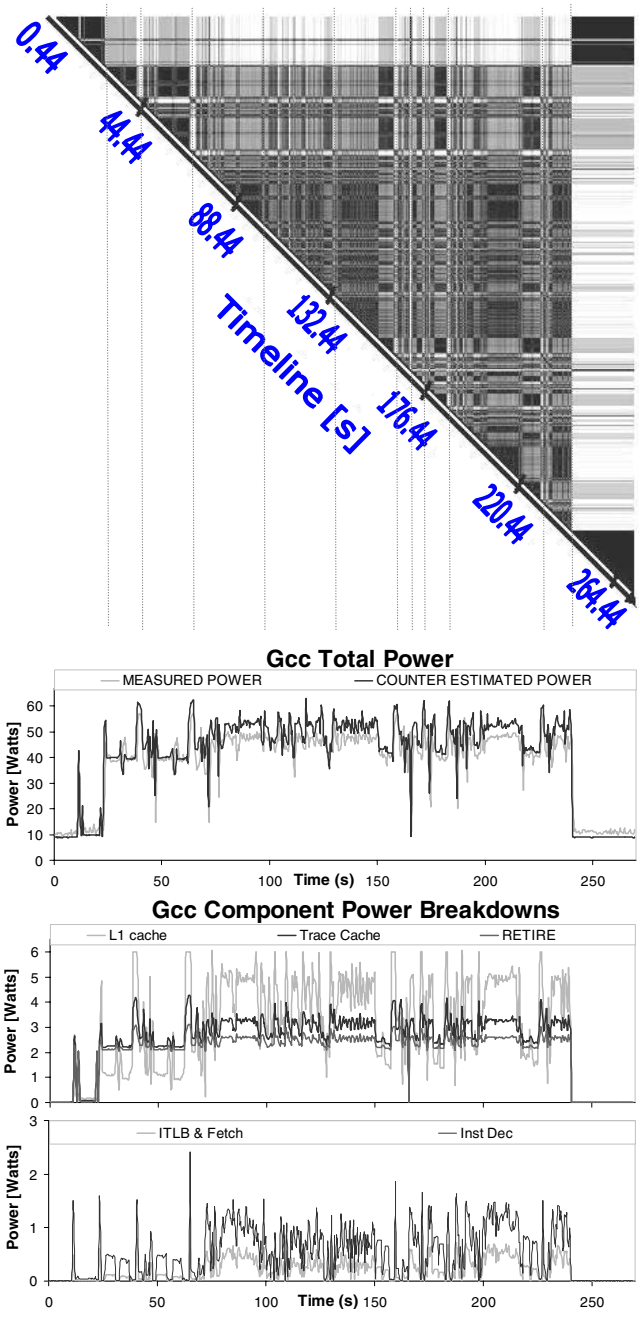


Figure 16. Gcc power phase similarity matrix.

(Vertical lines extending from the similarity matrix plots correspond to their projected time value in the power traces)

is to describe what event counters would work best if power measurement, instead of or in addition to performance measurement, were the design goal.

9. Conclusion and Future work

In this paper we present a runtime power modeling methodology based on using hardware performance counters to estimate component power breakdowns for the Intel Pentium 4 processor. Our total power estimates validate with good accuracy, despite P4's complex pipeline and aggressive clock gating. By using real power measurements to compare counter-estimated power against measured processor power, we have a real-time measurement and validation scheme that can be applied at runtime with almost no overhead or perturbation. We used our power model to measure long runs from the SPEC2000 suite and typical practical desktop applications. The power results show that our technique is able to track closely even very fine trends in program behavior. Because our technique has per-component power breakdowns, we can also get unit-by-unit power estimates. Furthermore, we can treat this "vector" of component power estimates as a power signature that can effectively distinguish power phase behavior based on simple similarity analysis.

This research differs from previous power estimation work in several aspects. Our model is targeted towards a complex high-performance processor with fine microarchitectural details and highly variable power behavior. Our power measurement technique is non-disruptive, and the LKM-based implementation is highly-portable. The component breakdowns we produce are based on physical entities co-located on chip, as opposed to conceptual groupings. As a result, these component breakdowns can offer a foundation for future thermal modeling research. The fact that detailed power data can be collected in real-time is also important for thermal research, since the large thermal time constants mandate long simulation runs. Our counter-based power model estimates even very low processor power accurately, by using both linear and piecewise linear combinations of event counts.

There are several key contributions of this work. The measurement and estimation technique itself is portable, and can offer a viable alternative to many of the power simulations currently guiding research evaluations. The component breakdowns offer sufficient detail to be useful on their own, and their properties as a power signature for power-aware phase analysis seem to be even more promising. In conclusion, this work offers both a measurement technique, as well as characterization data about common programs running on a widely-used platform. We feel it offers a promising alternative to purely simulation-based power research.

References

[1] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International Symposium on Microarchitecture*, pages 245–257, 2000.

[2] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, Jan. 1998.

[3] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of 9th ACM SIGOPS European Workshop*, September 2000.

[4] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conference*, pages 244–248, 2001.

[5] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, 2002.

[6] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, Feb. 1999.

[7] J. Haid, G. Kafer, C. Steger, R. Weiss, , W. Schogler, and M. Manninger. Run-time energy estimation in system-on-a-chip designs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2003.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal, First Quarter 2001*, 2001. developer.intel.com/technology/itj/.

[9] M. Huang, J. Renau, and J. Torrellas. Profile-Based Energy Reduction in High-Performance Processors. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[10] Intel Corp. *VTune^{TM} Performance Analyzer 1.1*. http://developer.intel.com/software/products/vtune/vlin/.

[11] Intel Corp. IA-32 Intel Arch. Software Developer's Manual, Vol. 3: System Programming Guide, 2002. developer.intel.com/design/pentium4/manuals/245472.htm.

[12] Intel Corp. Intel Pentium 4 and Intel Xeon Processor Opt. Ref. Man., 2002. developer.intel.com/design/Pentium4/manuals/248966.htm.

[13] Intel Corp. *Intel Pentium 4 Processor in the 423 pin package / Intel 850 chipset platform*, 2002. developer.intel.com/design/chipsets/designex/298245.htm.

[14] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[15] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. Technical report, Princeton University Electrical Eng. Dept., Sep 2003.

[16] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *International Symposium on Low Power Electronics and Design, pages 135–140*, 2001.

[17] I. Kadayif, T. Chinoda, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. vEC: virtual energy counters. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 28–31, 2001.

[18] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization, International Conference on Computer Design*, Sept. 2000.

[19] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded RISC processors. In *LCTES/OM*, pages 1–10, 2001.

[20] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of the International Conference on Computer Design*, October 1998.

[21] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *7th Annual Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2003.

[22] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[24] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[25] B. Sprunt. *Brink and Abyss Pentium 4 Performance Counter Tools For Linux*, Feb. 2002. www.eg.bucknell.edu/bsprunt/emon/brink_abyss.

[26] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[27] The Standard Performance Evaluation Corporation. SPEC CPU2000 Suite. http://www.specbench.org/osg/cpu2000/.

[28] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[29] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.

[30] M. T. Zhang. Powering Intel(r) Pentium(r) 4 generation processors. In *IEEE Electrical Performance of Electronic Packaging Conference*, pages 215–218, 2001.