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

blind review

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 can aid in evaluating full-system behavior, and are imperative for studies requiring execution times too long for simulation, such as long-term thermal analysis. Furthermore, as processors become more complex and employ increasingly aggressive dynamic power management techniques, measuring live power dissipation for a running system becomes more important. Likewise, estimating per-unit power dissipation has also become more challenging, due to complicated interactions between hardware sub-units.

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 offers live power breakdowns for 22 of the major CPU subunits, including the caches, branch prediction, renaming hardware and others. Because the tool measures a live, running system, it has negligible run-time overhead. In this paper, we present power dissipation timelines for minutes of full application run-time. Our programs studied include SPEC2000 as well as other desktop workload applications such as web browsing and text editing. Such measurements give insights into the long-term power and thermal behavior of realistic workloads. Overall, this paper demonstrates a power measurement methodology for current, high-performance processors, 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 and to evaluate tradeoffs in potential designs.

Unfortunately, cycle-level processor simulations are time-consuming, and are often vulnerable to concerns about accuracy. Particularly for thermal studies, very long simulations can be required, since there are long time constants associated with processors approaching equilibrium thermal operating points [22].

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, disks, and many other aspects of "real-world" behavior.

While live measurements gain value from their completeness, it can often be difficult to "zoom in" and discern how different subcomponents have contributed 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 [14, 15].

Prior counter-based energy tools have been geared towards previous-generation processors such as the Pentium Pro. Since these processors used little clock gating, they had minimal power variability 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:

1. We describe a detailed methodology for gathering live, per-unit power estimates based on hardware performance counters in complicated and aggressively-clock-gated microprocessors.

2. We present live total power measurements for SPEC benchmarks as well as some common desktop applications.

3. From the long-running power measurements we have obtained, we develop a method of power-oriented phase analysis using Bayesian similarity matrices, and we present results for several SPEC applications.

The remainder of this paper is structured as follows. Section 2 gives an overview of our performance counter and 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 analyzes program phase behavior based on measured power signatures. Finally, Section 8 discusses related work and Section 9 gives our conclusions and offers some 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 also use estimates based on performance counter data to produce per-unit power breakdowns. Figure 1 shows the basic flow we employ.

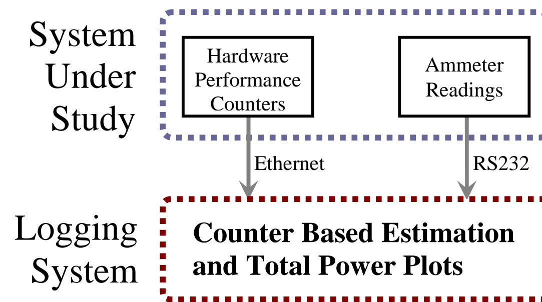


Figure 1. Overall power measurement and estimation system flow.

Live power measurements for the running system are obtained from an ammeter measuring current on the appropriate power lines. 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 [23, 10]. Mainly for efficiency and ease-of-use (as described in the next section) we have written our own Linux loadable kernel module (LKM) to access the counters. Our LKM-based implementation offers a mechanism with sufficient flexibility, while incurring almost zero power and performance overhead so that we can continuously collect counter information at runtime and generate runtime power statistics. Furthermore, the counter reader is easily installed on arbitrary Linux/P4 machines because of the LKM-based implementation.

The live power measurement for the Pentium 4 is obtained using a current probe—ammeter— to detect current drawn by the processor power lines. (While we could also probe power lines leading to disk and elsewhere, we choose to focus here on CPU behavior.) The clamp-style 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.

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. This is discussed further in Section 5. As the last step, we use the real power measurements obtained from the ammeter in conjunction with Power Client's counter-based power estimates to synchronize and provide a comparison between the measured and estimated total power measures.

Finally, we give some P4 implementation and microarchitectural details as an overview of the processor our research is based on. The machine measured in all power measurement and later power 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 processor implements extremely aggressive power management, clock gating, and thermal monitoring. Almost every processor unit is involved some kind of power reduction plan and almost every functional block contains clock gating logic, summing up to 350 unique clock gating conditions. With the help of this aggressive clock gating, P4 can provide up to approximately 20W power savings on typical applications [4].

The NetBurst microarchitecture is based on a 20-stage misprediction pipeline, that uses a trace cache that can hold up to 12K micro-ops (uops), which removes the instruction decoding from the main pipeline. In the NetBurst microarchitecture, a front-end BPU is used to predict branches fetched from an integrated 8-way 256kB L2 cache with 128 byte lines. A second smaller BPU is used to predict branches for uops within traces. The trace cache can issue 3 uops per cycle in deliver mode and 1 uop per cycle while building traces from instruction decoder. Two double-pumped ALUs are used for simple integer operations and the floating point unit is used for floating point and SIMD operations. The 4-way 8kB L1 cache with 64 byte lines is accessed in 2 cycles for integer loads, while the L2 cache is accessed in 7 cycles.

In the following sections we present our runtime power modeling methodology in this progressive manner. We first describe event monitoring with performance counters, then we describe the real power measurement setup and afterwards the power estimators we have developed.

3 Using Pentium 4 Performance Counters

Most of today's processors include some dedicated hardware performance counters and control, for debugging and event monitoring purposes. The general performance monitoring mechanism can be described in terms of several processing units generating various events, event detectors detecting these events and producing triggers for the counters, while accordingly configured counters incrementing with the corresponding triggers. A more

comprehensive description of event detection and counting can be found in [24].

Intel introduced performance counting into IA32 architecture with Pentium processors and has gone through two more counter generations since then. Pentium 4 performance counting hardware includes 18 hardware counters that can count 18 different events simultaneously 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. In addition to the 18 event counters, there exists a special time stamp counter (TSC) that increments with processor clock cycle ([11, 24]). 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 so forth [12].

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 which events should 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"

Because the performance counter has a simple and lightweight interface, we can completely control and update counters easily from within any application. (We can also run a helper application that reads counters at fixed intervals while another application-under-test runs.) The performance reader was written to be 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, because it requires copying data elsewhere in memory, but is invoked infrequently. For applications with moderate run-times where counter overflow is not yet an issue, get-event-counts need not be executed during the core of an application.

To validate the performance reader, we present counter data from microbenchmarks written to target specific processor units. The first benchmark, shown in Figure 2, generates a desired L1 cache hit rate by executing 1 billion iterations of traversing through a large linked list of pointers in a pseudorandom sequence with the degree of reuse guided by the user-specified desired hit rate. We use two metrics to evaluate the L1 hit rates. The first metric is to have 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

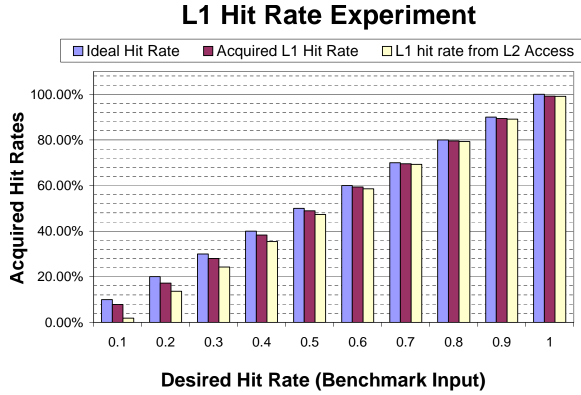


Figure 2. L1 hit rate experiment.

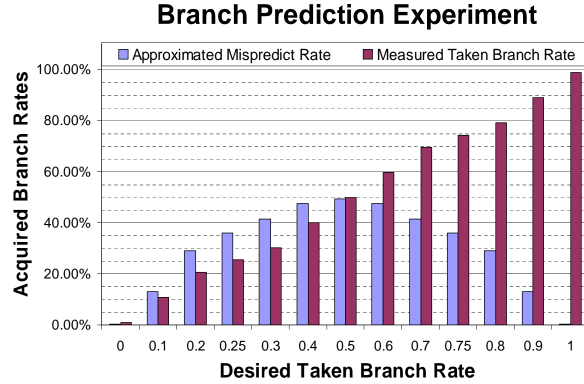


Figure 3. branch rate experiment.

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. While the counter-measured hit rates are both slightly below the ideal target, this is due to initialization effects in the microbenchmark code. In the main benchmark loop of 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

The live total power measurement in our setup (Figure 4) is accomplished via a clamp ammeter. While some prior power measurement work has added series or shunt resistors to the system to measure power dissipation [26, 19], we chose the clamp ammeter approach because it is unobtrusive and avoids needing to cut or add any wires on the system being measured.

The main power lines for the CPU operate at 12V, and then are fed to a voltage regulator module to convert this voltage to actual processor operating voltage and provide tight control over the voltage variations [28]. We verified this information by measuring the current through each of the 17 power lines in our system, while running a microbenchmark that creates high fluctuations in processor power. Three 12V lines yield current variations that

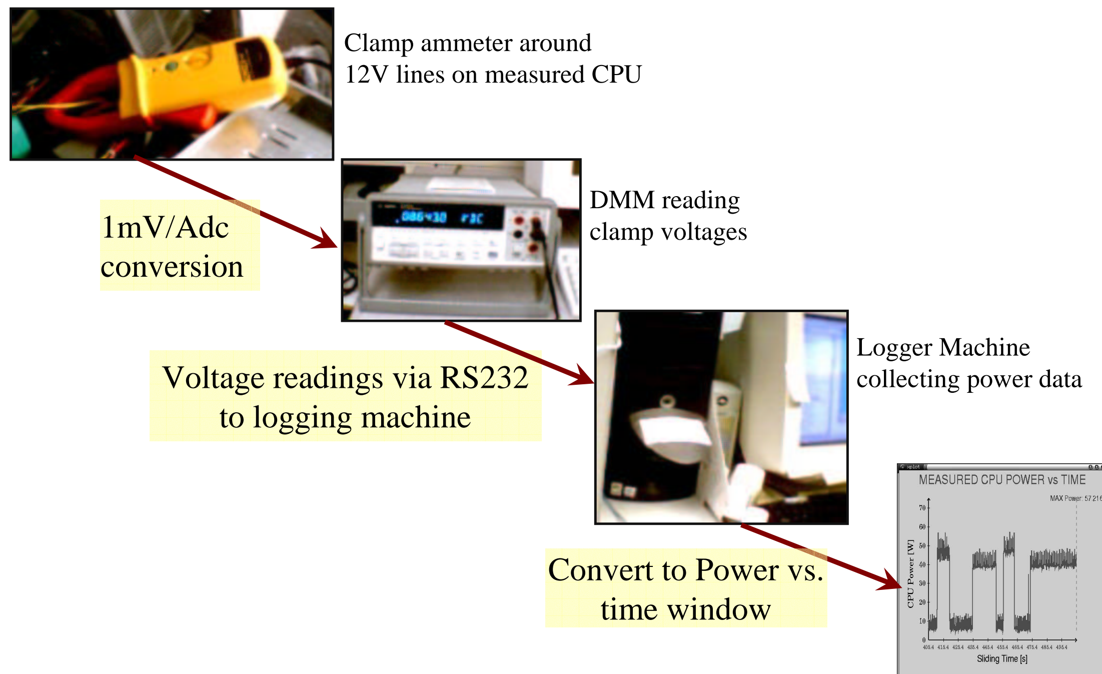


Figure 4. Processor Power Measurement Setup.

follow the processor activity tracked by the performance reader, while other lines present an uncorrelated power behavior, usually with insignificant power variation. Therefore, we use our current probe to measure the total current through the 12V lines.

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 the voltage readings from serial port and converts these values into processor power dissipation with the power relation: $P = V{\cdot}I = 12 \cdot (VoltageSample[V]){\cdot}1000$ and displays the measured runtime power in a *power monitor* over a running time window, while also logging the 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, therefore we collect the data in the logger machine at 20ms periods. 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.)

Before we show large-scale benchmark results in Section 6, we show here a snapshot of the runtime power monitor measuring the power of several microbenchmarks. In Figure 5, we show the power traces for four microbenchmarks. The leftmost benchmark, *fast*, is very simple code with two integer and one floating-point operation inside a computation-dominated loop. The next benchmark is the *branch microbenchmark* described in the

previous section. The third application in the timeline is called *hi-lo*, because it is an iterative stressmark designed to repeatedly swing power dissipation from very low to very high. Finally, the last three runs shown in the timeline correspond to the *cache microbenchmark* from the previous section, running with three different desired cache miss rates.

From the microbenchmark power traces, several characteristics are already clear. First, between applications, there is significant power variation 60W down to 25W depending on the program. Idle power further drops to around 8W. This power variation is due to partial utilizations of components and the resulting clock gating. Prior power measurement work, done for example on Pentium Pro processors, showed power variations of only a handful of Watts [14]. The hi-lo benchmark was particularly written to highlight this power swing; it has a roughly 30W swing as part of each iteration of its main loop. The three different cases of the cache microbenchmark also show interesting power behavior. Depending on a user-specified hit rate target, the same code, with 8 billion IA32 instructions, generates a significantly different hit rate and therefore different power profile, and execution time. In the three cases shown here, the first case has a high L1 hit rate, the second one has a low L1 hit rate but small enough footprint to avoid L2 misses, and the third one has a low L1 hit rate and a large footprint to incur L2 misses as well. In the third case, phase behavior is clear: the initialization phase has lower power, and the core execution phase has higher and less variable power.

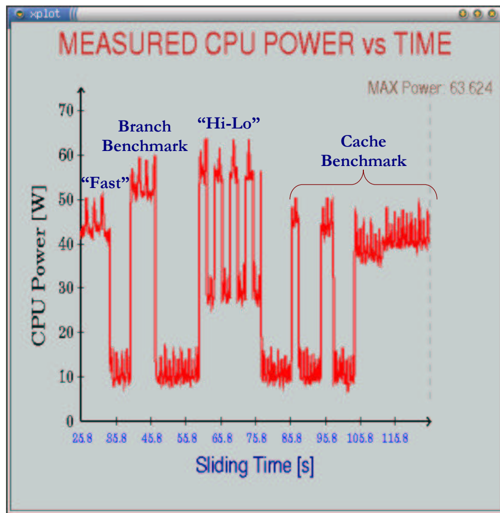


Figure 5. PowerPlotter Snapshot

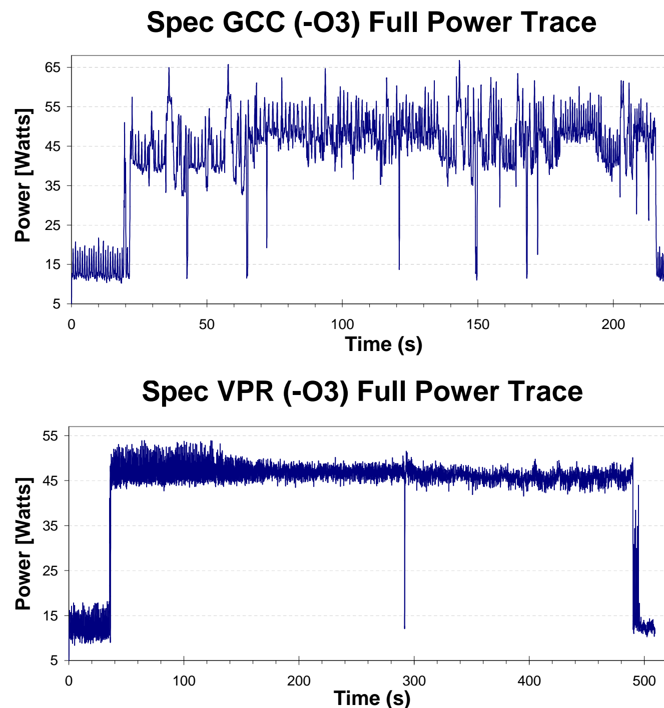


Figure 6. SPEC Power Examples.

We close this section with a selection of total power observations from the SPEC benchmarks. In Figure 6, 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 15, 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.

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 methods for this for much simpler processors [17, 3, 26, 14]. In our approach, we aim to estimate physical component powers using counter-based measures, and also to generate a reasonable total power estimate.

Our modeling technique presents distinct features compared to previous examples. 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. 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 as the components for which power should be reported. Or instead, one might use a more conceptual interpretation similar to [14] in which categories such as issue, execution, memory, fetch and reorder-related comprise the power breakdowns. 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 co-located 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 Defining P4 Events to Guide Estimation

Regarding the microarchitectural properties and available P4 events, we develop a first set of heuristics, which specify several P4 events, whose various combinations estimate the access rates for the defined components. The finalized set of heuristics involve 24 event metrics 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 performance counter metrics we devised to act as component access rates, which are used for power weighting.

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

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

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, which lies 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 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 total written speculative uops to get an integer estimated total. Integer operations that are not load/store and branch are scaled by 2 as they use double pumped ALU, while load/stores use address generation units and branch processing is done in complex ALU, together with shifts, flag logic and multiplies. We cannot differentiate multiply and shifts and therefore they are computed twice. Also, some x87 and SIMD instructions are decoded into multiple uops, which may cause undercounting.

At the end of all the metric definitions, we end up using 15 counters with 4 rotations. The P4 events and counter assignments are designed to minimize the amount of counter switches required to measure all the metrics needed. At least four rotations are unavoidable, as 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.

$$Power(C_i) = AccessRate(C_i) \cdot ArchitecturalScaling(C_i) \cdot MaxPower(C_i) + NonGatedClockPower(C_i) \quad (1)$$

Our estimation technique utilizes a piecewise linear relation between the access rates and component powers, specifically for the issue logic units. This emerges from our runtime observations, which reveal this sort of nonlinear behavior such that, an initial increase in from idle to a relatively low access rate for issue related components causes a large increase in processors power, while succeeding similar or higher amounts of increases in access rates produce much smaller power variations. Therefore, for the issue logic components, to be able to track this processor-awakening condition, we apply a conditional clock power factor in addition to linear scaling above a low threshold value that differentiates between gated and nongated clock conditions.

As an example of the overall technique, consider the trace cache. It delivers 3 uops/cycle when a trace is executed and builds one uop/cycle when instructions are being decoded into a trace. Therefore, the access rate approximation from 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 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 distributed into component power estimations.

$$Total\ Power = \sum_{i=0}^{22} Power(C_i) + Idle\ Power \quad (2)$$

For initial estimates of each component's "maxpower" value, we used physical areas on the die. Afterwards, we tuned our maximum power estimates and clock power assumptions based on the developed runtime verification

scheme for a small set of training benchmarks—the 4 microbenchmarks described in section 4—to achieve a much closer comparison between the modeled and measured total power as shown in figure 7, where the darker colored estimated power is plotted synchronously with the lighter colored measured power at runtime. Hence, we only used the shown 4 microbenchmarks to manipulate our estimates, in order to be able to objectively test our approximation with different programs. These four microbenchmarks are designed to produce stable processor behavior, with various simple execution characteristics, which helped us single out certain power behavior and test our corresponding estimations.

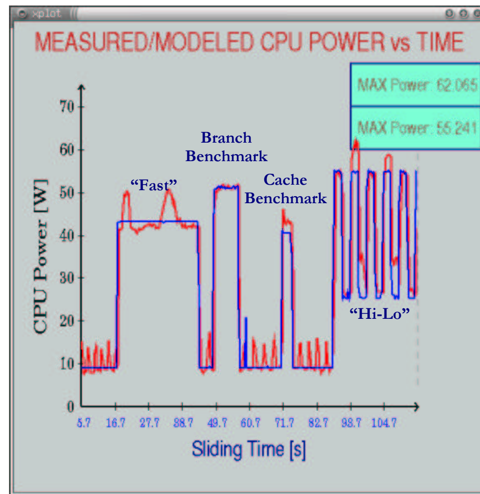


Figure 7. Total Power after tuning of maximum component powers and idle power assumptions

5.4 Final Implementation

Ultimately, our power modeling implementation is built upon the real power measurement setup shown in Figure 4, which is used for runtime verification. Additionally, 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.

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 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 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.

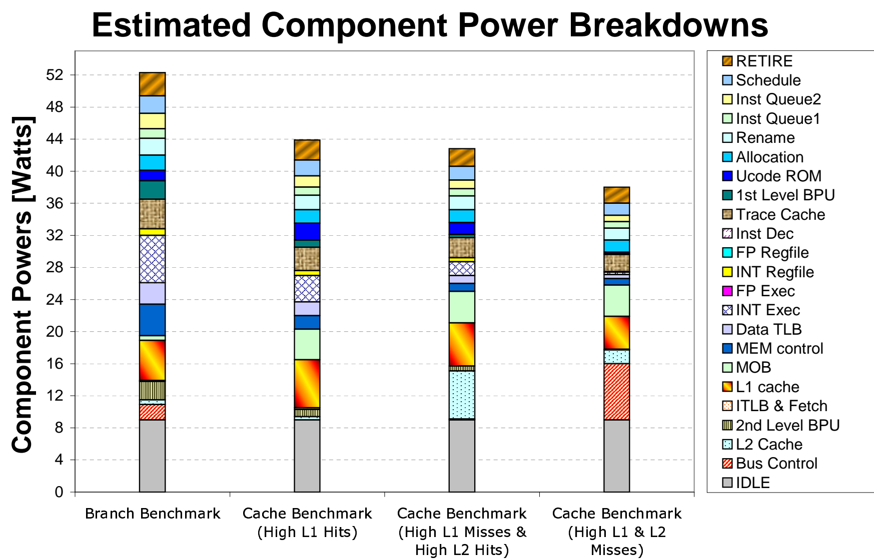


Figure 8. Power breakdowns for branch and cache benchmarks.

Leftmost bar of Figure 8 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 8 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 8, 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 8 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 as demonstrated in Figure 5, 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. 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, as well as equake from SPECfp (Power behaviors of gcc and gzip are referred to in section 7 and are not included here to avoid repetition.) In Figures 9–14, total power estimations and estimated power breakdowns for vpr, twolf and equake are demonstrated. Similar figures for gcc and gzip are included in Figures 18 and 19. We discuss the observed power behaviors of the demonstrated benchmarks and especially point out the cases, where component powers reveal power behavior that are not observable from total power measurements or estimations.

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 [25]. Although the total average power for the two runs is quite similar, Figure 9 shows a noticeable phase shift at around 300s when the second run begins and Figure 10 demonstrates even more clearly how distinct the power behavior in the second phase is. Although the first run, the placement algorithm, produces very stable powers, the second phase's routing algorithm has a much more variable

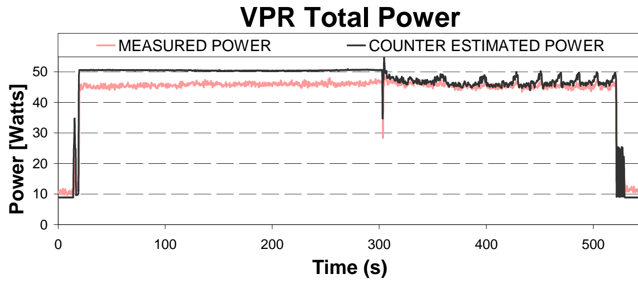


Figure 9. SPECint vpr Total Measured and Modeled Runtime Power.

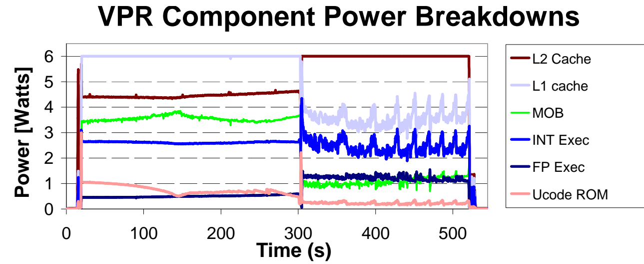


Figure 10. Estimated power breakdowns for VPR.

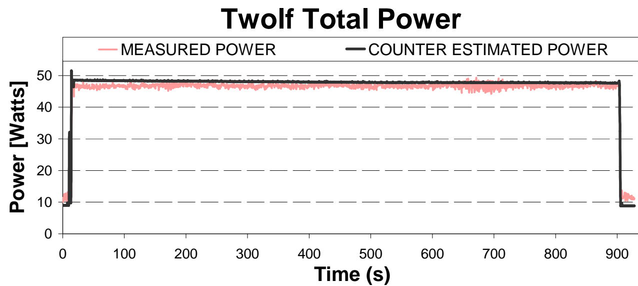


Figure 11. SPECint twolf Total Measured and Modeled Runtime Power.

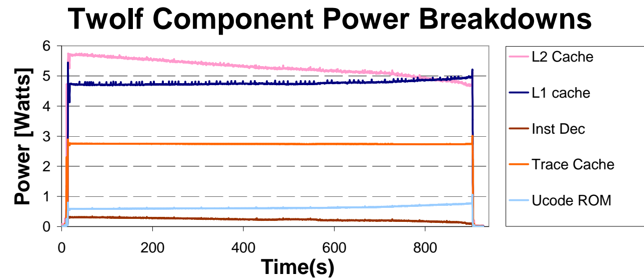


Figure 12. Estimated power breakdowns for twolf.

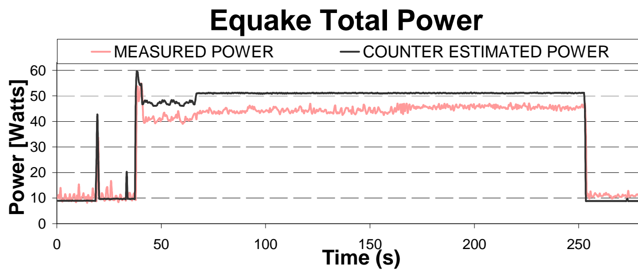


Figure 13. SPECfp equake Total Measured and Modeled Runtime Power.

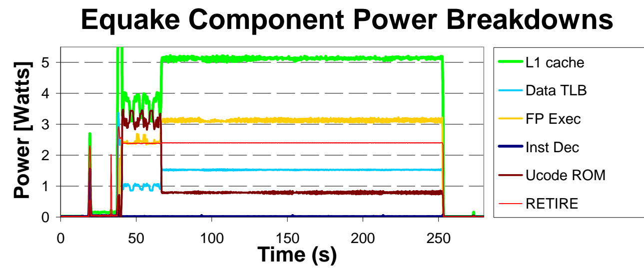


Figure 14. Estimated power breakdowns for equake.

and periodic power behavior. As [16] discuss, 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 show 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 the twolf is observed in the power breakdowns of Figure 12. Moreover, although twolf exhibits

an almost constant total power behavior in Figure 11, individual component powers generally show a gradient such as slight increases in L1 cache and microcode ROM powers and decreasing L2 cache power over the runtime.

As an example of floating point benchmarks, we show the equake benchmark in Figures 13 and 14. 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 13, we can already clearly identify the initialization phase and computation phase. Figure 14 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 and gzip are included in section 7 and the counter based power estimations are seen to track even the highly variant gcc power traces very favorably. Power traces for the rest of the investigated benchmarks are not included in this paper for brevity. In Figures 15 and 16, we demonstrate the statistical measures to represent the accuracy of our modeling framework, for a larger set of SPEC2000 benchmarks.

In Figures 15 and 16, 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. 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 13), 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 17, are AbiWord for text editing, Mozilla for web browsing and Gnumeric for numerical analysis.

In the web browsing experiment in Figure 17.(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

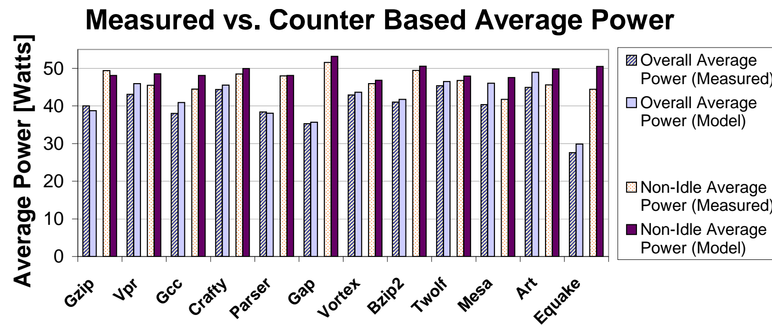


Figure 15. Average Power measurements and counter based estimations for SPEC2000 benchmarks. For each benchmark, first set of power values represent averaging over the whole runtime of the program, second set represents averaging only over non-idle periods.

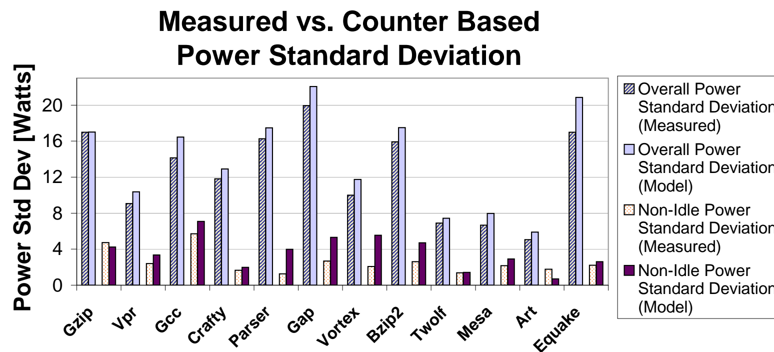
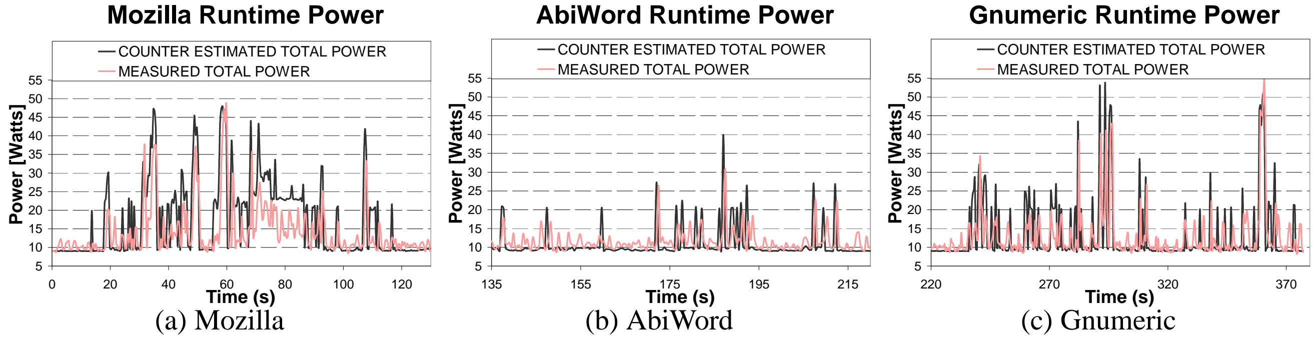


Figure 16. Variation of Power measurements and counter based estimations for SPEC2000 benchmarks. For each benchmark, first set of power values represent standard deviation for the whole runtime of the program, second set represents standard deviation for only non-idle periods.

17.(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 17.(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 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.



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

Figure 17. Total Measured and Counter Estimated Runtime Power for 3 desktop applications

7 Power Phase Behavior Analysis

Identifying program phases bears significant importance in computer architecture research. Several proposed algorithms implement specific techniques to detect these phases and provide feedback for microarchitecure. For example, [1] describes an algorithm that keeps track of specified performance metrics over a fixed instruction window to detect program phases for a multiconfigurable cache. Identification of phase behavior enables dynamic configuration of microarchitecture features for power or performance optimizations that respond to program behavior changes, such as configurable memory allocation and configurable instruction windows [5]. At larger scales of execution, phase behavior can be used for OS based dynamic management for thread scheduling, voltage or frequency scaling [9, 27]. Moreover, [21] demonstrates a methodology to approximate long costly architectural simulations using phase behavior to specify multiple simulation checkpoints.

As we have exemplified with several cases in power measurement and modeling sections, most programs exhibit different phases during their execution, usually including some level of cyclic behavior. In order to identify these periodic behaviors and standalone phases like initialization, [20] proposes *Basic Block Distribution Analysis* method that uses basic block profiles of programs. A basic block is a portion of a program code that is entered at one point, executed in whole and has a single exit point. The work in [20] introduces basic block vectors, which represent the proportion of basic block executions within one sampling period. Then, it uses basic block vector differences with respect to a global vector, to identify phases. In order to determine the amount of resemblance between different windows of execution, collected over the program run, [21] defines the basic block similarity matrix, which consists of the manhattan distance between all pairs of basic block vectors. The similarity matrix presents both the duration of similarities and the similar repetitions.

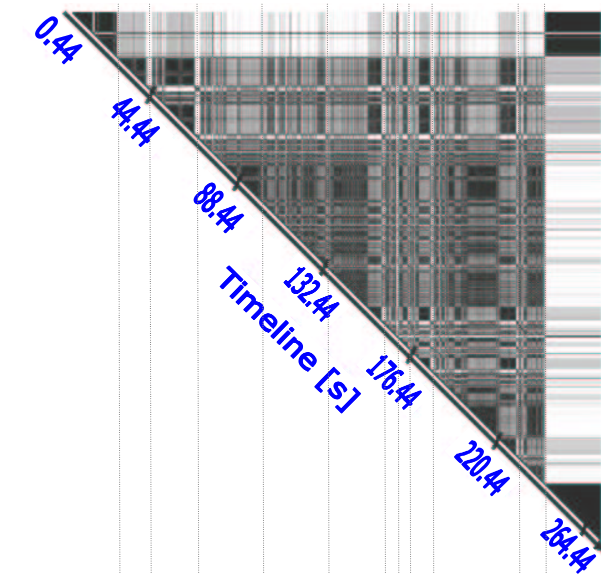
Here, we demonstrate a technique based on our runtime power model, to identify power phases of programs, and to use component breakdowns as characteristic power signatures. Because this technique works at runtime, it is quite efficient, and one can even imagine it being used to home in accurately on repetitive program phases even when the program source code is not available.

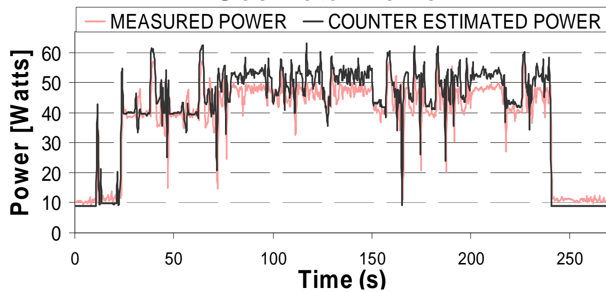
We consider generated component breakdowns as coordinate vectors in the space of possible. Proceedings similarly to the prior work on basic block vectors, we consider the manhattan distance between pairs of vectors as the "power behavior dissimilarity" between the two vectors. We do not normalize the power breakdown vectors, as our measure is already based on power and scaled power values should not be considered as similar power behavior. We rely on manhattan distance as the measure of dissimilarity, as it treats each change equally regardless of the direction of change. Also at any sampling time, the manhattan distance to origin represents the total power (except for the idle offset), at that sampling interval.

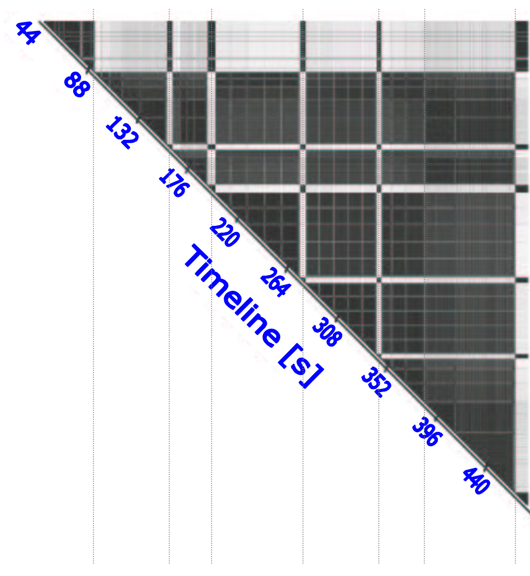
$$Similarity\ Matrix(r, c) = \sum_{i=0}^{22} |Power_r(C_i) - Power_c(C_i)| \tag{3}$$

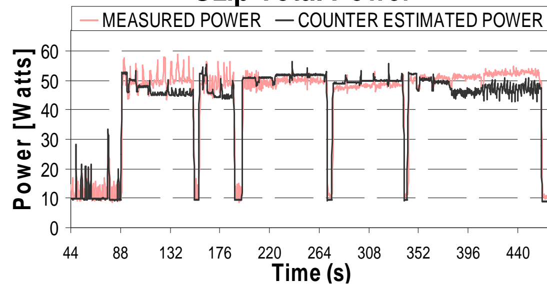
We use the similarity matrix approach to demonstrate the power phase behavior across the execution time of the programs. The similarity matrix is constructed from manhattan distances of all the combination pairs of component power vectors. A single matrix entry is computed as shown in equation 3, where $Power_{r,c}$ 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 display the similarity between the current time sample as defined by the diagonal entry and the future samples. The entries above the current sample show the similarity with respect to the previous samples.

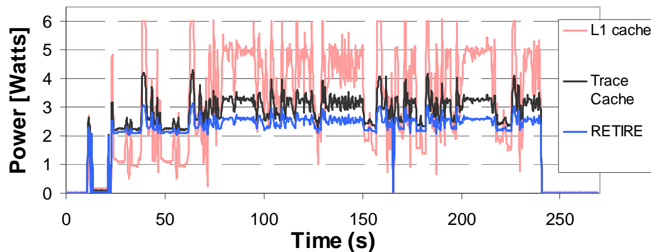
In Figures 18 and 19, we present the acquired power similarity matrices for SPEC2000 benchmarks gcc and gzip. In the matrix plots, the top left corner represents the start of the timeline and the lower right corner represents end of timeline. Darker regions represent higher similarity between the corresponding component power vectors. In the figures, we also show the total power and component power breakdown estimations for the benchmarks, with the same timescales to provide comparisons with the estimated power behavior. Even with gcc, with very unstable power behavior, several similarities are highlighted with the similarity matrix. For example, consider the almost identical power behavior around the 30, 50 and 180 seconds 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 88s, 210s and 230s regions show visibly higher similarity among themselves than the other two regions, which are also shown to be mutually dissimilar. On the other hand, gzip shows a much regular pattern with several similar phases. Once again, spurious similarities due to similar measured total power are distinguished with the similarity matrix, such as 100-150s and 200-280s regions. By providing a foundation for power phase analysis, counter-based component breakdowns have shown themselves to be useful in a range of power-aware and thermal-aware research studies.

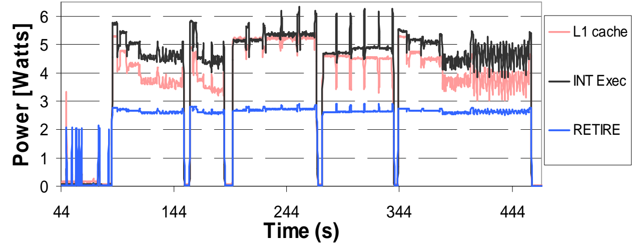


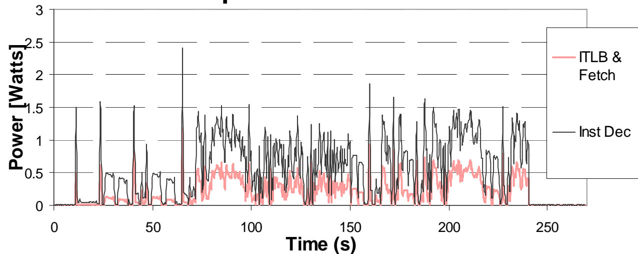












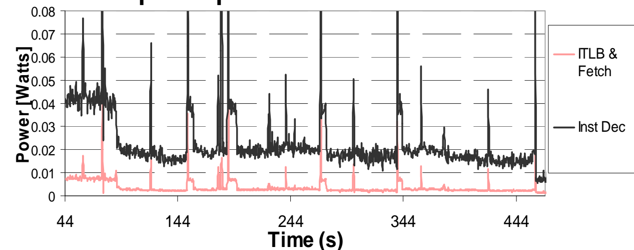


Figure 18. Gcc Power Phase Similarity Matrix

Figure 19. Gzip Power Phase Similarity Matrix

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

8 Related Work

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

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. [26] developed software power models for an Intel 486DX2 processor and DRAM and verified total power by measuring the current drawn while running programs. They used the information to generate instruction energy cost tables and identify inter-instruction effects. Russell et al. [18] likewise did software power modelling for i960 embedded processors, and validated using current measurements. Flinn et al. [6], developed the PowerScope tool, which maps consumed energy to program structure at procedural level. This OS-oriented research used a DMM to do live power measurements, and then developed energy analyzer software to attribute the power to different processes or procedures.

More recently, Lee et al. [17], used energy measurements based on charge transfer to derive instruction energy consumption models for a RISC ARM7TDMI processor. They used linear regression to fit the model equations to measured energy at each clock cycle. These techniques were aimed at very simple processors with almost no clock gating, however, and therefore needed 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 [19]. They use two series resistors in Vcc traces to measure the processor current. However, the scope of this work is only power measurement and they do not present any power breakdowns or power-oriented phase analysis.

Next, we present prior work on performance counters and power metrics. Bellosa [3], uses performance counters, to identify correlations between certain processor events, such as retired floating point operations, and energy consumption for an Intel PentiumII processor. This counter-based energy accounting scheme is proposed 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. [14], 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 needed to make 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. [15], use the Perfmon counter library to access performance counters of the UltraSPARC processor. 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 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 presented a runtime power modeling methodology based on using hardware performance counters to estimate component power breakdowns for the Intel Pentium 4 processor. We have validated our results at the total power level 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 acquired 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 breakdown, 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 very long simulation runs. Our counter-based power model estimating even very low processor power accurately, by using both linear and non-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 measurement technique, as well as characterization data about common programs running on a widely-used platform and 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. http://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 Corporation. *VTune^{TM} Performance Analyzer 1.1*. http://developer.intel.com/software/products/vtune/vlin/.

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

[12] Intel Corporation. Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, 2002. http://developer.intel.com/design/Pentium4/manuals/248966.htm.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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/.

[22] 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.

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

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

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

[26] 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.

[27] 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.

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