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

PRINCETON UNIVERSITY DEPT. OF ELECTRICAL ENGINEERING TECH. REPORT EE-2003-09*

SEPTEMBER 2003

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

Abstract

Power modeling and measurement are crucial elements for power aware research and there have been several attempts at varying levels of detail to address good and yet efficient modeling and measurement methodologies.

In our conference paper we introduced our live, runtime power estimation project and presented a synchronized real power measurement and power modeling technique at runtime for Intel Netburst architecture, P4-Willamette core implementation, using performance counters as a means to estimate component access rates. We used this access information for component power approximation.

This report provides the complete set of used access rate heuristics and expands the results presented in the conference paper to a larger group of benchmark applications.

Overall, we conclude that performance counter based component power estimation provides useful results for runtime power modeling and program power phase analysis regardless of any apriori program structure knowledge.

1 Introduction

As processor power densities keep their exponential trend [2], power and thermal issues present more stringent challenges due to battery life increasing with a slower pace and cost of heat removal systems increasing significantly above 60-70 ^{o}C. Therefore, a great deal of research effort focuses on power and thermally aware or adaptive systems in several levels of abstractions including software power profiling and compiler level power optimizations (for example [11, 10, 5, 8]), power modeling for power aware OS (for example [14, 13, 12]) down to microarchitectural level power estimation techniques [6].Most of these techniques need to rely on some amount of power measurement or modeling schemes that reflect modern processors, in order to quantify CPU power dissipation. In the ideal case, designers require detailed power breakdowns for processor units in order to assess realistic power measures, however these breakdowns are not available to acquire directly from the processors due to the lack of on chip 'energy meters', and simplistic measures such as constant maximum component power assumptions do not reflect modern processors' behavior because of microarchitectural power management mechanisms such as clock gating. In general, architects, OS and compiler developers either rely on timely simulations or resort to other metrics such as constant processor power [9], on/off processor

* This report includes the extensions to the paper appearing in the 36th International Symposium on Microarchitecture (MICRO-36), San Diego, CA, December 2003

power [14], instruction power profile based energy estimation [11, 8], IPC [1] as proxies for power measures.

Although simulators are pushed to architecture level sacrificing precision for computation speed, they are still not feasible to use for long timescale observations. In typical architectural simulators such as SimpleScalar[4] or Wattch [3], 1 second of real program run takes simulation times in the order of hours. Therefore, an important requirement for simulations of such characteristics is the provision of fast real-time modeling techniques. Moreover, dynamic thermal and power management require runtime modeling and measurement mechanisms to utilize in the dynamic decision making schemes.

At the bottomline, our project targets at estimating component based power breakdowns and processor temperature distribution at runtime, while providing real time verification based on real measurement, all regarding a Pentium 4 processor. We divide the overall work into two major phases: Runtime power modeling and runtime thermal modeling. In this paper, we present the complete first phase of *the* project, which is originally based upon similar ideas as the Castle project [7]. Our work for runtime component power modeling spans three distinct aspects. We use P4 performance counters to approximate component access rates, our modeling framework converts the approximated access rates into estimated component powers and our real power measurement setup provides measured total processor power for verification feedback. Our project provides the first Pentium 4 based power modeling scheme, with a piecewise linear approach due to conditional clock power offset and limited maximum power. It devises a strictly physical component based power model, rather than more abstract processor breakdowns, black box assumption [1] for total power estimation or software level power profiling [11, 8], to enable thermal modeling on top of our power model. Our power measurement setup produces no intervention to processor hardware and our performance counter reader implementation produces invisible power or performance overhead to actual system operation. Moreover, our power model aims at modeling low utilization powers accurately as high utilization powers, thus enabling us to model more practical applications such as web browsing and text editing with reasonable accuracy.

2 Related Work

There is significant amount of work focusing on processor power modeling and measurement, as well as application of these modeling techniques to power aware systems. Here we discuss work that falls into one of two categories: First, research related to processor power measurement, and its application to power modeling; second, research that employs performance counters for power metrics. In the earlier work, Tiwari et al. [11], measure the current for an Intel 486DX2 processor and DRAM to generate instruction energy cost tables and identify inter-instruction effects such as circuit state overhead and resource constraint effects. Russell et al. [9], model software power for two implementations of i960 embedded processor architecture by a simple constant average power estimation. They consider a program's runtime as the only determination factor in program energy consumption and use a series resistance to measure processor current for energy verification. Flinn et al. [5], developed PowerScope tool, which maps consumed energy to program structure at procedural level. They sample current drawn by the processor with a DMM and an energy analyzer software generates procedure power profiles from the raw energy data. Lee et al. [5], uses a cycle based energy consumption measurement system based on charge transfer to derive instruction energy consumption models for a RISC ARM7TDMI processor. They follow a black box approach similar to [1] to model total processor power based on linear regression to fit the model equation to measured energy consumption at each clock cycle. As a first example of Pentium 4 power measurement, Seng et al. [15], investigate the effect of compiler optimizations on average program power, by measuring the processor power for

benchmarks compiled with different optimization levels. They use two series resistors in Vcc traces to measure the processor current.

Regarding employment of performance counters for power metrics, Bellosa [1], 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. Castle project, developed by Joseph et al. [7], 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. Kadayif et al. [16], use 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. Haid et al. [6], propose a coprocessor for runtime energy estimation for system-on-a-chip designs. The developed *JouleDoc* coprocessor estimates component energy consumptions at runtime with special event counters in conjunction with power macromodels.

3 Modeling Power for P4 Sub-Units Details

In our conference paper we have described how we estimate component-wise powers for the P4 processor at runtime using the available event counting metrics. We described the steps involved in the estimation framework as first defining the components from annotated die layout, which decomposed the processor into 22 different units. We emphasize, the choice of physical components as the modeled units in turn enables one to build a thermal model on top of the power estimation framework in a structured manner. Afterwards, we discussed some of the heuristics that we used, to approximate the access rates for these components, which consequently serve as the proxies to component power dissipations. Next, we described the applied empirical method to convert these access rates into component power consumption estimates, with the help of tuning benchmarks and described the actual implementation.

In the following subsections, we include some additional information related to our power modeling framework that were not included in the paper due to space limitations. We demonstrate the experimental setup, provide more details on performance metrics and tuned power estimates.

3.1 Experimental Setup for Power Estimation and Concurrent Verification

In Figure 1, we demonstrate the overall setup for power estimation and verification with respect to total power. The left branch over the Ethernet represents the performance counter data collection from the machine under test for power estimation. The right branch shows the power measurement flow from the current probe over the power lines to the digital multimeter and from the multimeter to the logger machine. The leftside monitor is the runtime total power monitor that displays total measured vs. estimated power over a 100s sliding window. The rightside monitor shows the component breakdowns at each instant.

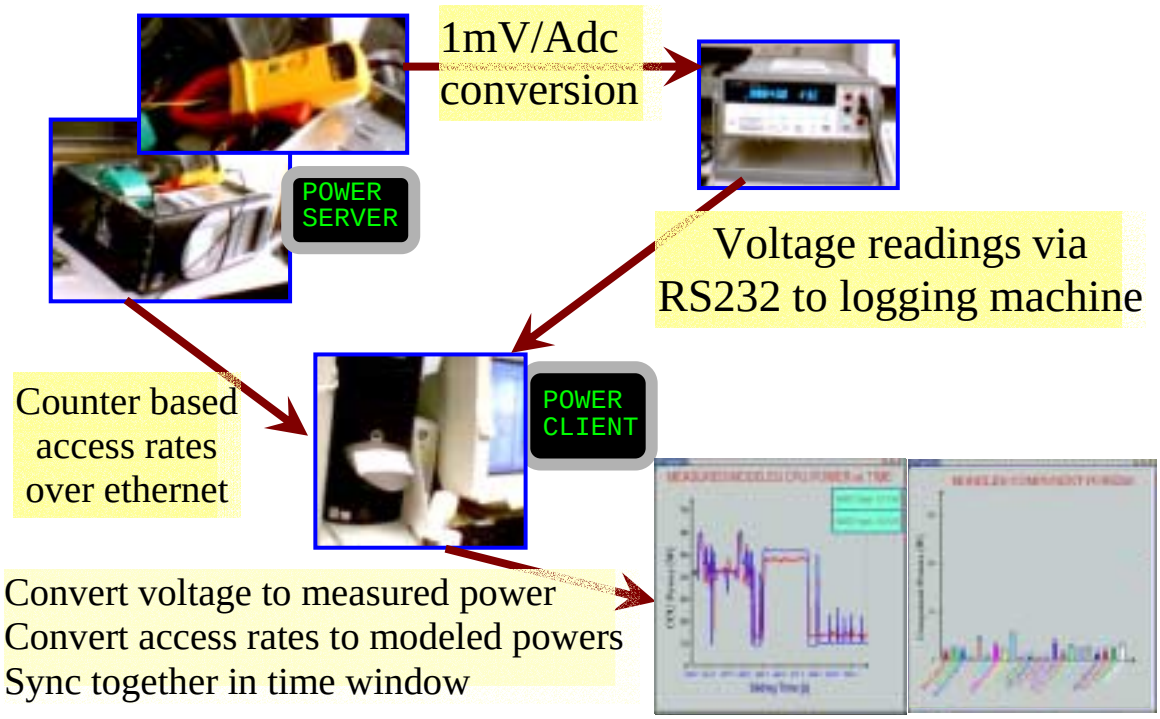

Figure 1, Complete Experimental Setup

3.2 Complete Set of Event Metrics for Component Access Rate Estimation

Although we present some of the performance metrics in the original paper, the complete list was too long to include. Here we present the used metric combinations for each of the modeled components as well as brief descriptions of masks and scalings. For each component, the final metrics are used as representatives of access rates.

1) Bus Control:

As there is no 3rd Level cache we assume Bus Sequence Queue (BSQ) allocations are similar to Input Output Queue (IOQ) allocations. Regarding Figure 2, we devise two metrics to account for BSQ and IOQ, and to account for driving the actual backside bus. The first used metric is bus accesses from all agents, which go to the IOQ. The used event is "IOQ_allocation", which counts various types of bus transactions. We assume this metric should account for BSQ as well, and therefore do not include a second "BSQ_allocation" metric. The allocation metrics are access based rather than duration. The used event mask is "0x0EFE1", which corresponds to: "Default req. type, all read (128B) and write (64B) types, include OWN,OTHER and PREFETCH. The second metric is bus utilization (The % of time Bus is utilized). The used event is "FSB_data_activity", which counts DataReaDY and DataBuSY events on Bus. The mask is "0x03F' and makes the counter count when processor or other agents drive/read/reserve the bus. The expression for bus utilization then is "FSB_data_activity x BusRatio / Clocks Elapsed", where bus ratio is used to account for clock ratios between bus and processor core.

Final access rate is represented as:

$$\frac{IOQ\ Allocation}{\Delta Cycles_1} + \frac{Bus\ Ratio \cdot FSB\ Data\ Activity}{\Delta Cycles_2}$$

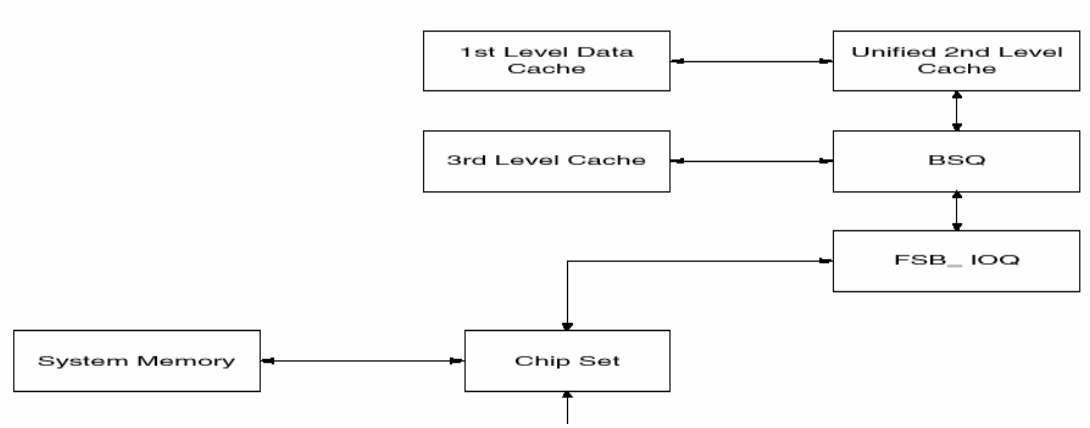

Figure 2, Memory subsystem.

2) L2 Cache:

The metric used is 2nd level cache references. The event used is "BSQ_cache_reference", which counts cache references as seen by the bus unit. The mask is: "0x0507" and corresponds to all MESI reads (LD & RFO) and 2nd level WR misses.

Final Expression is:

$$\frac{BSQ\ Cache\ Ref}{\Delta Cycles_1}$$

3) 2nd Level BPU:

We use two metrics. First metric is instructions fetched from L2, as instructions are not decoded yet, they need to go to the BPU to check if they are branches for the least and need to be predicted if they are. The event used is "ITLB_Reference", which counts ITLB translations. The mask is "0x01" and corresponds to all hits. The expression we use for instructions fetched is "8xITLB_Reference" as there are minimum 8 instructions per L2 cache line. The L2 cache line is 128 bytes and the maximum instruction length for IA32 instructions is 16 bytes. The second metric is "Branches retired" as the retirement logic does the history update at these instances for the front end BPU. The event used id "branch_retired", which counts branches retired. The mask is: "0x0F" to count all Taken/NT/Predicted/Mispredicted.

The Final expression is:

$$\frac{8 \cdot ITLB\ Reference}{\Delta Cycles_1} + \frac{Branch\ Retired}{\Delta Cycles_2}$$

4) ITLB & I-Fetch:

We use two metrics. First metric is ITLB translations performed, for the ITLB accesses. The event used is "ITLB_Reference", which counts ITLB translations. The mask is "0x07" and accounts for all hits and misses. The second metric is Intruction fetch requests by the front end BPU, for the fetch unit. The event used is "BPU_fetch_requests", which counts Ifetch requests from the BPU. The mask is "0x01", which is TC lookup misses. This is all that is documented for the time being as the available option for counting

Final expression is:

$$\frac{ITLB\ Ref}{\Delta Cycles_1} + \frac{BPU\ Fetch\ Req}{\Delta Cycles_2}$$

5) L1 Cache:

We use two metrics. First metric is loads and stores retired that successfully completed data speculations, and possible replays. The event used is "Front End Event", which counts tagged uops that retired. The mask is "0x03" to count both non-speculative (NBOGUS) and speculatives (BOGUS). There is a supporting event for front end tagging, named "Uop type", which can tag types of instructions – Load and Store instructions for our case. The mask is "0x06" to tag both Loads and Stores. The second metric is replays for extra accesses due to data speculation. The used events are "LD port replay", which counts replayed events at load port with mask "0x02", Split LD (all that is available for now), and "ST port replay" (same as "Memory Complete", with mask: SSC), which counts replayed events at store port. The mask for this event is "0x02" to count Split ST (all that is available for now).

Final expression is:

$$\frac{Ld\ Port\ Replay + St\ Port\ Replay}{\Delta Cycles_1} + \frac{Front\ End\ Event}{\Delta Cycles_2}$$

6) MOB:

There is no metric that we could devise for MOB accesses directly. Therefore the metric used is LD Replays triggered by MOB. The event used is "MOB load replay" that counts the load operations replayed by MOB. The mask is "0x03A" and corresponds to all replays due to unknown address/data, partial data match, misaligned addresses.

Final expression is:

$$\frac{MOB\ Load\ Replay}{\Delta Cycles_2}$$

7) Memory Control:

We cannot find any relevant metric for memory controller accesses and make the conservative assumption that memory controller dissipates maximum power as long as machine is not idle. The metric we use to make this discriminations is non-idle clock cycles and the event is "Machine Clear", which counts cycles when the pipeline is flushed. The mask is "0x01" and counts machine clears due to any cause. The expression for non-idle clock cycles is "TSC count – Machine Clear Cycles".

Final Expression is:

$$1 - \left(\frac{MachineClearCycles}{\Delta Cycles_2}\right)$$

8) DTLB:

The metric is accesses to either L1 or to MOB and therefore the total TLB access rates are considered as the total of L1 and MOB accesses. The expression is: "L1 Accesses + MOB Accesses"

The final expression is:

$$L1\ Cache\ Access\ Rate + MOB\ Access\ Rate$$

9) FP Execution:

There is no aggregate metric for floating point operations, and therefore we use 8 different metrics to collect all the floating point execution information. Our metric is FP instructions executed and the events are:

- event1: “packed_SP_uop”, which counts packed single precision uops
- event2: “packed_DP_uop”, which counts packed double precision uops
- event3: “scalar_SP_uop”, which counts scalar single precision uops
- event4: “scalar_DP_uop”, which counts scalar double precision uops
- event5: “64bit_MMX_uop”, which counts MMX uops with 64bit SIMD operands
- event6: “128bit_MMX_uop”, which counts integer SSE2 uops with 128bit SIMD operands
- event7: “x87_FP_UOP”, which counts x87 FP uops
- event8: “x87_SIMD_moves_uop”, which counts x87, FP, MMX, SSE, SSE2 ld/st/mov uops

The masks for events 1-7 are “0x08000” to count all, which is the only available option. The mask for 8th event is “0x018” to count SIMD moves/stores/loads.

The final expression is:

$$\left(\frac{PackedSPuops+PackedDPuops}{\Delta Cycles_1}\right)+\left(\frac{ScalarSPuops+ScalarDPuops}{\Delta Cycles_2}\right)+\left(\frac{64bitMMXuops+128bitMMXuops}{\Delta Cycles_3}\right)+\left(\frac{x87FPuops+x87SIMDmovesuops}{\Delta Cycles_4}\right)$$

10) Integer Execution:

The desired metric for integer execution is integer uops executed, however there is no associated event with integer execution. Therefore we use a substitute metric as total speculative Uops executed and subtract the FP uops computed above. The used event is “Uop queue writes”, which counts number of uops written to the uop queue in front of TC. The mask is “0x07” to counts all uops from TC, Decoder and Microcode ROM. The expression for integer execution is: “Uop Rate – FP uop rate”, with some postfix for simple vs. complex ALU operations as described in our conference paper.

In the hardcoded expression we actually I rescale FP uop rates as packed,SIMD and MMX uops do multiple concurrent FP operations. Moreover, there is a final fix to avoid negative integer counts. However, a simpler demonstration for final expression is:

$$2 \cdot \left(\frac{Uop\ Queue\ Writes}{\Delta Cycles_1} - FP\ Exe.\ Access\ Rate\right) - L1\ Cache\ Access\ Rate - \frac{Branch\ Retired}{\Delta Cycles_2}$$

11) Integer Regfile

The metric we use is integer uops executed, as there is no direct metric for total physical regfile accesses.

Final expression is:

Integer Execution Access Rate

12) FP Regfile

The used metric is FP uops executed as there is no direct metric for total physical regfile accesses.

The final expression is:

Floating Point Execution Access Rate

13) Instruction Decode:

The used metric is cycles spent in trace building. The event used is “TC Deliver Mode”, whichc counts the cycles processor spends in the specified mode. The mask is “0x038” to count the cycles logical processor 0 is in build mode.

Final expression is:

$$\frac{TC\_DeliverMode}{\Delta Cycles_1}$$

14) Trace Cache:

The used metric is uop queue writes from either trace cache modes. The event used is "Uop queue writes", which counts number of uops written to the uop queue in front of TC. The mask is "0x07" to count all uops from TC and Decoder and ROM.

Final expression is:

$$\frac{Uop\ Queue\ Writes}{\Delta Cycles_1}$$

15) 1st Level BPU:

The used metric is branches retired. The event used is "branch_retired", which counts branches retired. The mask is "0x0F" to count all Taken / Not Taken / Predicted / MisPredicted branches.

Final expression is

$$\frac{Branch\ Retired}{\Delta Cycles_2}$$:

16) Microcode ROM:

The used metric is uops originating from microcode ROM. The event used is: "Uop queue writes", which counts number of uops written to the uop queue in front of TC. The mask is "0x04" to count uops only from microcode ROM.

Final expression is:

$$\frac{UopQueueWrites_{(0x04)}}{\Delta Cycles_1}$$

17) Allocation, 18) Rename logic, 19) Instruction queue1,
20) Schedule, 21) Instruction queue2:

For all these issue related components, we cannot devise separate metrics that can differentiate among them. Therefore, we use the common metric of uops that started their flight for all of them. The used event is: "Uop queue writes", which counts number of uops written to the uop queue. The mask: is "0x07" to count all uops from TC and Decoder and ROM.

Final expression is:

$$\frac{Uop\ Queue\ Writes}{\Delta Cycles_1}$$

22) Retirement Logic:

The obvious metric to is uops that arrive retirement. The used event is: "uops retired", which counts number of uops retired in a cycle. The mask is "0x03" to consider also speculative uops retired.

Final expression is:

$$\frac{UopsRetired}{\Delta Cycles_1}$$

3.3 Used Counters, Rotations and Corresponding ESCRs

Counters	Rotation1	Rotation2	Rotation3	Rotation4
cntr0	IOQ_allocation	IOQ_allocation	FSB_data_activity	FSB_data_activity
cntr1	BSQ_cache_ref	BSQ_cache_ref	BSQ_cache_ref	BSQ_cache_ref
cntr2	BPU_fetch_rqsts	BPU_fetch_rqsts	MOB_ld_replay	MOB_ld_replay
cntr3	ITLB_reference	ITLB_reference	ITLB_reference	ITLB_reference
cntr4	uop_queue_writes(0x07)	uop_queue_writes(0x07)	uop_queue_writes(0x07)	uop_queue_writes(0x07)
cntr5	TC_deliver_mode	TC_deliver_mode	TC_deliver_mode	TC_deliver_mode
cntr6	uop_queue_writes(0x04)	uop_queue_writes(0x04)	uop_queue_writes(0x04)	uop_queue_writes(0x04)
cntr7	-	-	-	-
cntr8	packed_SP_uop	scalar_SP_uop	64bit_MMX_uop	x87_FP_uop
cntr9	LD_port_replay	LD_port_replay	LD_port_replay	LD_port_replay
cntr10	packed_DP_uop	scalar_DP_uop	128bit_MMX_uop	x87_SIMD_moves_uop
cntr11	ST_port_replay	ST_port_replay	ST_port_replay	ST_port_replay
cntr12	branch_retired	branch_retired	machine_clear	machine_clear
cntr13	uops_retired	uops_retired	uops_retired	uops_retired
cntr14	front_end_event	front_end_event	front_end_event	front_end_event
cntr15	uop_type	uop_type	uop_type	uop_type
cntr16	-	-	-	-
cntr17	-	-	-	-

3.4 Power Numbers

Unit	Area %	Area Based Max Power Estimate	Max Power after Tuning	Conditional Clk power
L2 BPU	3.4%	2.5	15.5	-
L1 BPU	4.9%	3.5	10.5	
Tr. Cache	8.6%	6.2	4.0	2.0
L1 Cache	5.8%	4.2	12.4 (/2)	
L2 Cache	14.7%	10.6	300.6(/7)	
Int EXE	2.0%	1.4	3.4	
FP EXE	6.2%	4.5	4.5	
Rename	2.3%	1.7	0.4	1.5
Retire	6.5%	4.7(/3)	0.5	2.0

4 Extended Results

5 Conclusion and Future Work

In this paper we presented a runtime power modeling methodology based on hardware performance counters as proxies for processor component access rates, which, in turn, are used to estimate component power breakdowns for the processor. In our implementation we used a Pentium 4 processor that utilizes the highest extent of clock gating for power management among present modern processors. We used real power measurements to verify modeled total processor power against measured real processor power, thus providing a real validation scheme that can also be applied at runtime. We evaluated our power model with several examples from Spec2000 suite to typical practical desktop applications and the acquired estimation results verified our proposed model follows even very fine trends in program behavior within good approximation for a runtime scheme based on a very complex processor. We also described an application to the implemented power model that can be used to 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 very complex high performance processor with fine microarchitectural details and highly variable power behavior due to employed aggressive clock gating and power management techniques. We utilize a power measurement technique that does not disrupt the processor hardware and a LKM based performance monitoring scheme that induces invisible power and performance overhead on the measured system. Both of these aspects make the modeling scheme very powrtable and easily repeatable. We rely on a strictly physical component based power breakdowns, with fine granularity, which enables implementation of a processor thermal model that can work in conjunction with the power model. Developed power model aims at estimating even low processor utilization powers accurately, thus providing reliable power values for any range of operation power. Therefore, we use a nonlinear power model with conditional component clock powers, unlike linear regression based examples that try to fit a model to sampled measured powers for different programs that generally produce significant processor utilization.

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.

The most major prospect of the developed power model is the development of a thermal model that works in conjunction with the component power estimations. Realization of thermal modeling had been the driving reason to use physical layout based components in the first place. Another future direction that we aim to follow is investigation and evaluation of different estimation techniques for better power estimation. There are already examples that utilize linear regression of raw counter metrics to fit measured power for a set of training programs. However, we aim to investigate techniques that will produce power estimations that track processor powers in any range of power, while also preserving the component level breakdown information. Eventually, we aim to provide a combined runtime power and thermal modeling framework, with runtime verification, that estimates processor power and temperature distributions accurately; and to use it in future power and thermal related research.

References

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

[2] S. Borkar. Design challenges of technology scaling. *IEEE Micr*o, 19(4):23–29, July 1999.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architectur*e, June 2000.

[4] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture New*s, pages 13–25, June 1997.

[5] 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 Application*s, pages 2–10, Feb. 1999.

[6] J. Haid, G. Kfer, C. Steger, R. Weiss, , W. Schgler, 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.

[7] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Interna-tional Symposium on Low Power Electronics and Design, pages 135–14*0, 2001.

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

[9] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embed-ded processors. In *Proceedings of the International Conference on Computer Desig*n, October 1998.

[10] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the IEEE Symposium on Low-Power Electronic*s, October 1994.

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

[12] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the first USENIX Symposium on Operating Systems Design and Implementatio*n, pages 13–23, Nov. 1994.

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

[14] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X*), Oct. 2002.

[15] 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.
[16] 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.