Watts-inside: A Hardware-Software Cooperative Approach for Multicore Power Debugging

Jie Chen, Fan Yao, Guru Venkataramani
Department of Electrical and Computer Engineering,
The George Washington University, Washington DC, USA
{*jiec, albertyao, guruv*}*@gwu.edu*

***Abstract*—Multicore computing presents unique challenges for performance and power optimizations due to the multiplicity of cores and the complexity of interactions between the hardware resources. Understanding multicore power and its implications on application behavior is critical to the future of multicore software development. In this paper, we propose Watts-inside, a hardware-software cooperative framework that relies on the efficiency of hardware support to accurately gather application power profiles, and utilizes software support and causation principles for a more comprehensive understanding of application power. We show the design of our framework, along with certain optimizations that increase the ease of implementation. We present a case study using two real applications, Ocean (Splash-2) and Streamcluster (Parsec-1.0) where, with the help of feedback from Watts-inside framework, we made simple code modifications and realized up to 5% power savings on chip power consumption.**

Keywords*—*Power Debugging, Multicore Power, Hardware-Software Cooperative Framework

I. Introduction

With the limitations posed by Dennard scaling, power-related issues grow significantly in future multicore chip designs and ultimately limit the scalability of multicore computing [6]. There is also an increasing need to understand power consumption at the application level such that programmers and compilers can deploy static code optimizations without having to rely on expensive runtime power saving strategies.

Conventional power saving strategies utilize dynamic, hardware-based solutions such as Dynamic Voltage-Frequency Scaling, Power and Clock Gating. Unfortunately, most of such mechanisms can be cost-ineffective on applications that are *not statically tuned for power*. On the other hand, software-only power profiling tools are mostly disadvantaged by their limited knowledge of the underlying hardware parameters and the inability to calibrate power of hardware functional units with reasonable accuracy. Therefore, a more effective strategy is to combine the hardware capability of providing an accurate view of the program behavior with the software flexibility to effect low-cost, program-level power optimizations.

In this paper, we explore *Watts-inside*, a novel hardware-software cooperative solution framework for *Multicore Power Debugging* that detects opportunities for power optimization. Our goal is to provide feedback on the power consumption of applications at a finer-grain level such that the programmers and compilers can effect power-related optimizations on program code regions. We utilize hardware support to profile power for program code sequences[1], and include additional hardware to efficiently identify the functional unit behind higher power consumption. We then use software support and probability of causation principles [14] to understand application power at a finer granularity, such that we can attribute the cause for high power to a short sequence of instructions. We note that such a framework can play a vital role in the future of multicore software development by assisting programmers and compilers with useful suggestions on which code regions can take advantage of power optimizations.

The contributions of our work are as follows:

- We motivate the need for power debugging, especially for multicores, and explore a hardware-software cooperative framework to analyze application power. We identify fine-grain power-related bottlenecks and attribute them to short sequences of program code.
- We design efficient hardware mechanisms that use filtering (removing certain uninteresting code sequences from further hardware-level analysis), and sampling (minimizing the number of code sequences to be analyzed) to reduce the impact on application performance.
- We apply *probability of causation* principles to estimate the degree to which a particular finer-grain code block (say, instructions within a basic block) could be the reason behind higher power consumption measured in the code sequence.
- We propose and evaluate our designs, the resulting cost and complexity using Splash-2 [22] and PARSEC-1.0 [2] benchmarks.
- We present a case study in Section VI-E, where we show how our framework can assist in identifying and improving power in a couple of real-world applications.

II. Motivation – Understanding Multicore Power

To understand the power consumption behavior of applications, we perform experiments that characterize their power when executing on symmetric multicore processors. We note that more complex multicore environments that are asymmetric or heterogeneous can present even further challenges. In our studies, we run four-threaded applications on four core

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557.

[1]Since it is impossible to measure power at the level of basic blocks containing just a few instructions, we consider dynamic sequences of N basic blocks for which we estimate average power using power proxy modules that are already available in many modern processors [8], [18].

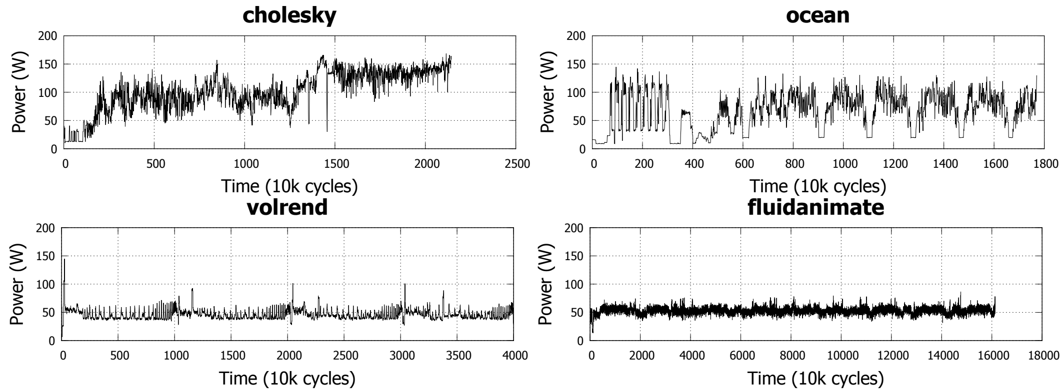

Fig. 1. Dynamic Power traces for four-threaded applications measured during their execution

TABLE I. PERFORMANCE AND DYNAMIC POWER IMBALANCE IN SPLASH-2 AND PARSEC-1.0 BENCHMARKS WITH FOUR THREADS

Application	Parallel Section (File/Function)	Num. of Dynamic Instances	% of Appln. Exec. Time	Avg. Performance Imbalance	Avg. Dynamic Power Imbalance
Volrend (SPLASH-2)	adaptive.c/ray_trace(...)	3	44.33%	0.001%	10.21%
Barnes (SPLASH-2)	load.C/maketree(...)	4	72.47%	0.90%	8.89%
Cholesky (SPLASH-2)	solve.C/Go(...)	1	28.09%	2.82%	31.74%
Bodytrack (PARSEC-1.0)	WorkerGroup.cpp/ WorkerGroup::Run()	82	50.06%	0.47%	4.32%

processors without placing any specific constraints on power consumption or voltage-frequency settings, i.e., the settings are assumed to result in the best possible execution time. We measure chipwide power during intervals of 10,000 cycles by running our benchmarks on SESC [17], a cycle-accurate architecture simulator with an integrated dynamic power model that uses Wattch [4] and Cacti [12] for power estimation[2]. 32 nm technology is assumed in all of our experiments. Figure 1 shows dynamic power traces for a representative subset of our benchmark applications when executing on four-core processors. Our results indicate that different multicore applications can exhibit different characteristics during the various phases of their execution– (1) monotonously increasing power, e.g., cholesky, (2) phases of high and low power, e.g., ocean, (3) occasional peaks of high power, e.g., volrend, and (4) almost uniform power, e.g., fluidanimate.

Even for applications that have been thoroughly debugged for performance and load balanced, our studies show that the parallel sections of multicore applications could still suffer from uneven power consumption between multiple cores. Table I shows parallel sections in some of the well-known Splash-2 and Parsec-1.0 applications that are running on four cores with four threads and shows the average imbalance[3] in performance and power across several dynamic instances of the parallel section. Despite almost perfect performance balance that can be achieved through hardware optimizations like out-of-order execution and prefetching, we see significant power imbalance (up to 31.7% in cholesky) across the different cores because power consumption by the functional units are still determined by the amount of work to be done. These results are consistent with a recent survey by Chen et al [5] and *show the necessity to understand the application's power characteristics in greater detail in order to accurately effect changes that improve power consumption.*

[2]We note that recent proposals like McPAT [13] can perform more accurate modeling based on technology projections from the ITRS roadmap [16], but our simulation infrastructure does not currently support McPAT and we are working on upgrading our framework.

[3]Average power (or performance) imbalance in an application's parallel section is measured as the average difference between the threads having the highest and lowest power (or execution time).

III. WATTS-INSIDE: A FRAMEWORK FOR DEBUGGING MULTICORE POWER

A. Hardware Support

To improve power, the user (programmer, compiler or the hardware) should first understand which parts of the program code suffer from power-related issues and what functional units are responsible for this effect. We design hardware support that estimates dynamic power for a string of N consecutively executing basic blocks (which we call Code Sequence), and log its power information in memory for further analysis. A code sequence is chosen as a granularity in our hardware design to capture meaningful power information that is relatable back to program code, while minimizing the hardware implementation complexity. In our experiments, we assume N=5 because it offers a nice trade-off between capturing power information at finer granularity and accuracy of power measurement on overlapped instructions. Sometimes, a code sequence can contain fewer than N basic blocks in cases of a function call/return and exceptions; we terminate such code sequences prematurely to prevent them from straddling program function boundaries and exceptions.

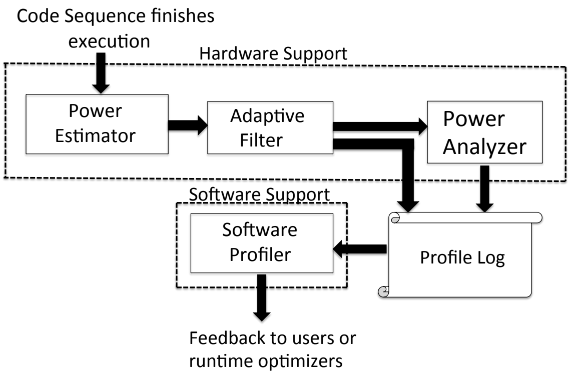

Fig. 2. Design overview of the Watts-inside framework

Figure 2 depicts an overview of our hardware-software design for Watts-inside framework. Conceptually, we divide the hardware support for Watts-inside into three stages, namely,

1) Power Estimator: This module is responsible for computing (or estimating) the power consumption of code sequences. The processor chip is embedded with activity sense points inside various functional units which are monitored by a power estimator unit. In our design, this module is conceptually similar to the IBM Power7's power proxy module that has specifically architected and programmably weighted counter-based architecture to keep track of activities and form an aggregate value denoting power [8].

2) Adaptive Filter: This module is responsible for filtering code sequences that are essentially 'uninteresting' with respect to power and do not warrant a *second hardware-level* analysis for functional unit-specific power information. Note that, when needed, the software profiler has the capability to analyze all code sequences regardless of filtering.

The adaptive filter has two active parameters– (1) maximum power so far (observed from the start of the application execution), and (2) capture ratio (C), a *user-defined* parameter that specifies the threshold for code sequences whose average power fall within the top $C\%$ of highest power (e.g., if the capture ratio is 10% and the highest power for any code sequence so far is 50 W, then the filter forwards all of the code sequences whose power consumption is at least 45 W.). *In the remaining sections of this paper, we refer to high power code sequences as ones within the capture ratio, and the remaining code sequences as low power (or NOT high power) for simplicity.*

To detect the high power sequences, the filter checks whether the code sequence (Q's) power falls within or exceeds the high power range. If true, then the filter forwards Q to the Power Analyzer for further processing. Whenever Q's power exceeds the maximum power observed thus far in the application, maximum power and threshold are updated. We note that after the maximum power reaches a stable value (i.e., after the highest power consuming sequence has executed), updates are no longer necessary.

3) Power Analyzer: This module is responsible for estimating the contribution of individual microarchitectural (or functional) units for high power code sequences, and then determining the functional unit that was responsible for the highest amount of power. We forward the output of this stage to a log that can be further analyzed by software profilers.

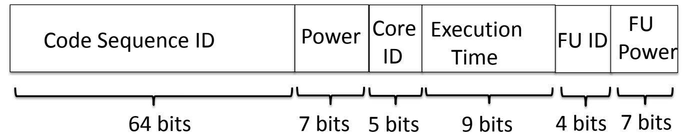

Fig. 3. Code Sequence Power Profile Vector (CSPPV)

For design efficiency, we adopt common activity based component power estimation that can estimate power for a large number of functional units using just a few generic performance counters [15]. We identify fourteen functional units (Instruction and Data Translation Lookaside Buffers, Instruction and Data Level-1 caches, Branch Predictor, Rename logic, Reorder Buffer, Register File, Scheduler, Integer ALU, Float ALU, Level-2 cache, Level-3 cache and Load Store Queue) to study the power breakdown by individual units . We chose these fourteen units based on our analysis of functional unit-level power consumption across our benchmark suites.

Figure 3 shows the output of the Watts-inside hardware. For each code sequence, we construct a 96-bit long Code Sequence Power Profile Vector (CSPPV) that includes:

- Code Sequence ID: The power estimator generates a unique 64-bit identifier for every code sequence by folding the 32-bit address of the first basic block, and then concatenating lower order bits of other constituent basic blocks within the code sequence.
- Code Sequence Power: The power estimator uses 7 bits to store the code sequence power.
- Core ID: 5 bits are used for the core ID where the code sequence executed, filled by power estimator.
- Execution time: The power estimator uses 9 bits to store the execution time of the code sequence. This can be later used for: (1) computing energy, and (2) ranking code sequences to prioritize longer running blocks.
- FU ID: The power analyzer uses 4 bits to uniquely identify the one of the fourteen functional units that consumes the most power.
- FU Power: The power analyzer uses 7 bits to show power consumed by the highest power consuming functional unit.

The power analyzer module records the CSPPV into a memory log that can later be utilized by software profilers.

B. Software Support

1) Causation probability: To help programmers and compilers apply targeted power-related optimizations to program code, feedback must be given at the level of fine-grain code blocks (say, a few instructions within a basic block). Toward this goal, we develop a causation probability model to determine whether an individual basic block within a code sequence could cause higher power.

Watts-inside quantifies the impact of a certain basic block B on the power of the code sequence Q using three probability metrics:

- Probability of Sufficiency (PS): If B is present, then Q consumes high power. A higher range of PS values indicate that the presence of B is a sufficient cause for Q's high power consumption.
- Probability of Necessity (PN): Among Qs that consume high power, if B were not present, then Q would have not consumed high power. A higher range of PN values indicate that the absence of B would have caused Q to lower its power.
- Probability of Necessity and Sufficiency (PNS): B's presence is both sufficient and necessary to infer that Q consumes high power. Higher values of PNS prove that B's likeliness to be the reason behind Q's higher power.

To compute the boundaries of PS, PN and PNS, we define the following additional probability terms:

Let b be the event that a basic block B occurs in a code sequence, and h be the event that the code sequence consumes high power. $P(h_b)$ denotes counterfactual relationship between b and h, i.e., the probability that if b had occurred, h would have been true.

$$P(h) = (\#HighPowerSeq)/(\#Seq) \quad (1)$$
$$P(b, h) = (\#HighPowerSeqWithB)/(\#Seq) \quad (2)$$
$$P(b', h') = (\#LowPowerSeqWithoutB)/(\#Seq) \quad (3)$$
$$P(h_b) = (\#HighPowerSeqWithB)/(\#SeqWithB) \quad (4)$$
$$P(h_{b'}) = (\#HighPowerSeqWithoutB)/(\#SeqWithoutB) \quad (5)$$
$$P(h'_{b'}) = (\#LowPowerSeqWithoutB)/(\#SeqWithoutB) \quad (6)$$

The boundary values for PS, PN and PNS are defined below:

$$max\left\{0, \frac{P(h_b) - P(h)}{P(b', h')}\right\} \leq PS \leq min\left\{1, \frac{P(h_b) - P(b, h)}{P(b', h')}\right\} \quad (7)$$

$$max\left\{0, \frac{P(h) - P(h_{b'})}{P(b, h)}\right\} \leq PN \leq min\left\{1, \frac{P(h'_{b'}) - P(b', h')}{P(b, h)}\right\} \quad (8)$$

$$PNS \geq max\left\{\begin{array}{l} 0, P(h_b) - P(h_{b'}), \\ P(h) - P(h_{b'}), P(h_b) - P(h) \end{array}\right\} \quad (9)$$

$$PNS \leq min\left\{\begin{array}{l} P(h_b), P(h'_{b'}), P(b, h) + P(b', h'), \\ P(h_b) - P(h_{b'}) + P(b, h') + P(b', h) \end{array}\right\} \quad (10)$$

By using the boundary equations 7- 10, we present a few test cases below to verify our causation model:

• If a basic block B appears frequently in high power code sequences and sparsely in low power sequences, both PS and PN boundary values are very high (closer to 1.0). Consequently, PNS values are also very high. Such blocks *are certainly* candidates for power optimization. For example, if there are 1000 code sequences, of which 200 are classified as high power (via capture ratio C). Let us assume that B_1 appears in 100 of the high power code sequences, and does not appear in any low power sequences. Using the boundary equations, we find that $1 \leq PS \leq 1$ and $0.9 \leq PN \leq 1$. These high PS and PN values show that B_1 is certainly a candidate for power optimizations.

• If a basic block B appears sparsely in high power code sequences, both PS boundary values are closer to 0.0, and the PN boundary values are either a widely varying range or are closer to 0.0. Such blocks *cannot* be good candidates for power improvement considerations. Using the same example above, let us assume that the block B_2 appears in 5 of the 200 high power code sequences and B_2 appears in 95 of the 800 low power (NOT high power) code sequences. Using the boundary equations, we find that $0 \leq PS \leq 0.06$ and $0 \leq PN \leq 1$. Low PS values combined with practically unbounded PN values indicate that B_2 cannot be a good candidate for power improvement.

• If a basic block B appears $L\%$ of the time in high power code sequences and $M\%$ of the time in low power sequences (where L and M are non-trivial), PNS boundary values determine the degree to which B's likeliness in causing higher power in the corresponding program code sequences. Therefore, higher ranges of PNS values for B indicates higher benefit in applying power-related optimizations to B. Using the example described above, let us consider two blocks B_3 and B_4– (1) B_3 appears in 40 of the 200 high power code sequences and in 200 of the 800 low power code sequences, where $0 \leq PNS \leq 0.167$, and (2) B_4 appears in 35 of the 200 high power sequences and 20 of the 800 low power sequences, where $0.462 \leq PNS \leq 0.636$. Even though B_3 appears more frequently in high power code sequences than B_4, there is higher benefit to optimizing B_4 because of its *larger* high power causation probability.

We find that this approach mathematically helps us to quantify the degree to which a specific set of instructions result in higher power consumption.

2) Code Sequences with varying power consumption between cores: Our software support can improve the quality of feedback information via two mechanisms – (1) Use clustering algorithms (e.g., k-means) to cluster sequences based on the degree of power variation, i.e., code sequences that show higher power variation are clustered separately from the ones that have lower power variation. This can aid runtime systems to do better scheduling of threads and map them on to cores that satisfy their power needs. (2) Identify the cause for power variation using the CSPPVs. Since the vector contains information on functional unit consuming the highest power, it can facilitate targeted optimizations including code changes and dynamic recompilation.

3) Predicting potential for Thermal Hotspots: By monitoring a contiguous stream of code sequences executing on the same core where a functional unit repeatedly contributes to the highest portion of power, we could predict parts of the chip where thermal hotspots could develop. Also, by having information on the physical chip floorplan, we can even detect local thermal hotspots resulting out of continuously high activity in adjacent functional units. Such analysis can effectively help temperature-aware software development of multicore applications.

IV. Implementation

In this section, we show how our framework can be integrated with a modern multicore architecture.

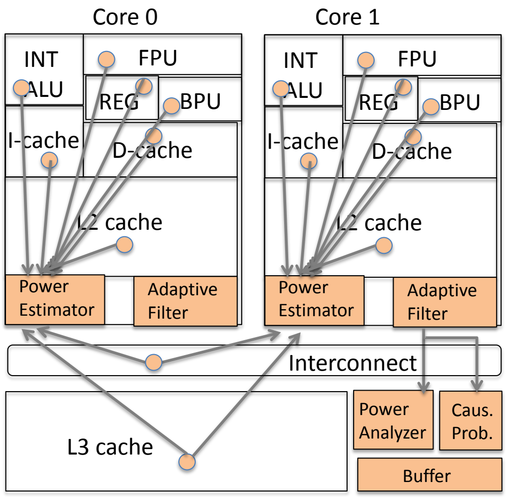

Fig. 4. Hardware modifications needed for Watts-inside framework

A. Hardware Support

Figure 4 shows the hardware modifications needed to implement Watts-inside framework. We include the power estimator (similar to the modules found in modern processors like Intel Sandybridge, IBM Power7) and adaptive filter logic locally in every core. After power estimation, our adaptive filter determines whether this block warrants further processing. To do this, there are two special registers- a programmable register to store the user-desired capture ratio, and an internal register to hold maximum power observed so far.

We implement the power analyzer module as a centralized resource that is shared by all cores within a multicore chip. The adaptive filters inside the cores forward only the high power code sequences to the power analyzer.

To reduce the performance impact of hardware profiling, we consider two more optimizations– (1) hardware buffer to accumulate the CSPPVs and update memory when the bus is idle, and (2) sampling of code sequences to minimize the impact on multicore performance.

Additionally, we implement an online hardware causation probability module and a watch register (that can be programmed by the user with a specific basic block address). This is conceptually similar to setting watchpoints in program debuggers. The adaptive filter forwards all of the code sequences that contains the *basic block address under watch* to the hardware causation probability module, that in turn computes the PS, PN and PNS values. We believe that such a feature shall aid runtime systems like dynamic recompilation or adaptive schedulers to optimize specific code regions during program execution.

B. Software Support

We run the software profiler as a separate privileged process in the kernel mode. The profiler supports APIs for functions such as (1) querying which basic blocks have high power causation probability (note: this offline software implementation is more comprehensive and separate from the online hardware causation probability module in Section IV-A), (2)

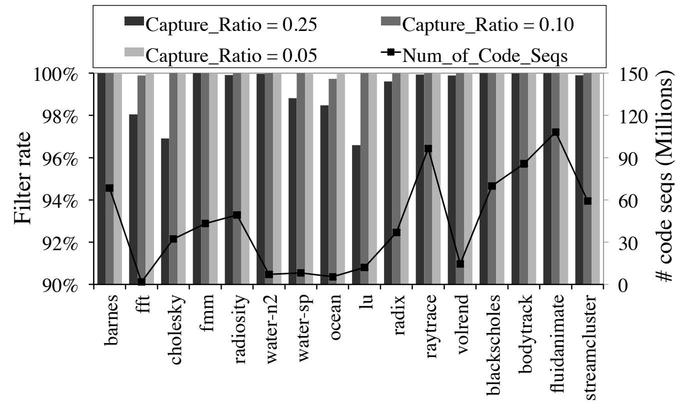

Fig. 5. Ideal Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads.

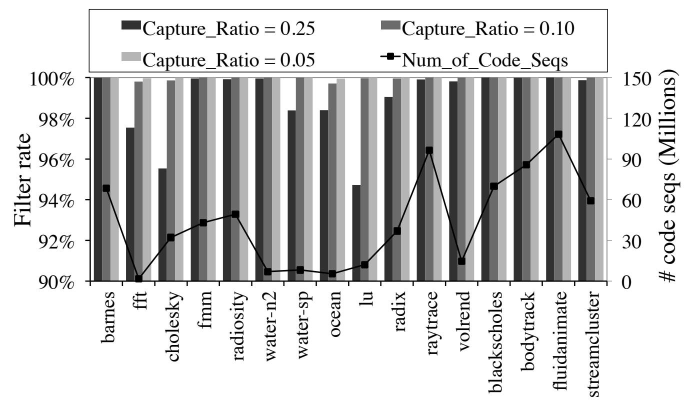

Fig. 6. Adaptive Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads.

automatically mining the CSPPVs for basic blocks that cause higher power. This software profiler gets its input from the CSPPV log created by the power analyzer. The memory pages belonging to CSPPV log are managed by the Operating System and are allocated on demand. If the OS senses that memory demands of CSPPV log interferes with the performance of regular applications, the OS pre-emptively deallocates certain memory pages and/or alter the sampling rate of code sequences to minimize the memory demands of CSPPV log. Also, we use Lempel-Ziv coding to compress and decompress CSPPV logs [23], that helps us to reduce memory footprint sizes.

V. Experimental Setup

We use SESC, a cycle accurate architectural multicore simulator [17] that has an integrated power model. We model a four-core Intel Core i7-like processor [9] running at 3 GHz, 4-way, out-of-order core, each with a private 32 KB, 8-way associative Level 1 cache and a private 256 KB, 8-way associative Level 2 cache. All cores share a 16 MB, 16-way set-associative Level 3 cache. The Level 2 caches are kept coherent using the MESI protocol. The block size is assumed to be 64 Bytes in all caches. We use parallel applications from Splash-2 [22] and PARSEC-1.0 [2] that were compiled by gcc with -O3 flag, and run four-threaded version on four cores.

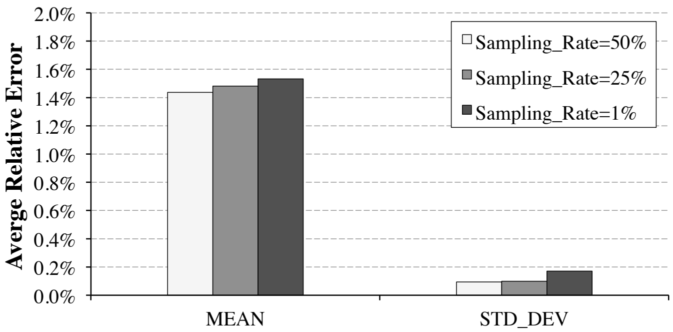

Fig. 7. Average relative error in the mean and standard deviation of power consumption by code sequences (in Splash-2 and Parsec-1.0 applications) due to sampling, relative to a baseline execution without sampling.

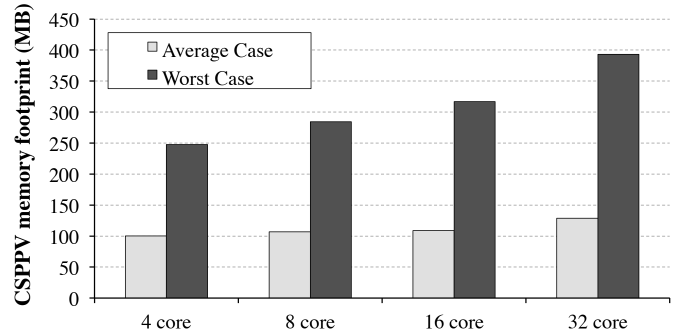

Fig. 8. Average and worst-case CSPPV Memory log requirements for capture ratio=0.25 for different numbers of cores

VI. Evaluation

A. Adaptive Filter vs. Ideal Filter

In this experiment, we compare the effectiveness of our adaptive filter (that *adjusts its threshold dynamically* to filter code sequences based on the maximum power seen thus far and the capture ratio) against an ideal filter (that *does not need to adjust the threshold dynamically* because it has prior knowledge of the maximum power consumed by any code sequence in the multicore application and the capture ratio). Figures 5 and 6 show the results of our experiments. For each benchmark, we show the percentage of code sequences that are filtered for three separate capture ratios namely 0.25 (or code sequences within top 25% of maximum power), 0.10 and 0.05 respectively. On the right axis, we show the total number of code sequences that are executed by each application. As an example, cholesky benchmark executes a total of 32.13 million code sequences; at a capture ratio of 0.25, 96.9% of the code sequences are filtered by ideal filter and 95.5% of the code sequences are filtered by adaptive filter.

Based on our experiments, we notice that in a majority of benchmarks, except fft, cholesky and lu, our adaptive filter successfully filters above 99% of the code sequences (for all three capture ratios) and sends only $\leq 1\%$ code sequences to the power analyzer module for further analysis. These filter rates are nearly same as that of the ideal filter. In fluidanimate that has a large number of code sequences, our adaptive filter performs nearly equal to the ideal filter in minimizing the number of sequences that are sent to the Power Analyzer. In certain benchmarks like lu, our adaptive design filters up to 3.8% less than an ideal filter, especially for capture ratio of 0.25. However, lu has fewer than 12 million code sequences and the absolute numbers of code sequences that reach power analysis stage are still far fewer than the benchmarks with hundreds of millions of code sequences. Therefore, we conclude that our adaptive filter design proves effective and is able to perform very close to an ideal filter.

B. Sampling

Even after filtering, for certain applications, the number of CSPPVs might still be high enough to cause significant performance overheads. To minimize the traffic of code sequences that reach the power analyzer module from various cores, we perform periodic sampling, i.e, one out of every N code sequences is chosen by Watts-inside framework for power estimation and analysis. Figure 7 shows the results of our experiments when we sample code sequences at the rates of 50%, 25% and 1%, and compare the observed mean and standard deviation of code sequence power with the baseline execution where we do not have sampling. At 50%, we note that periodic sampling introduces fairly low relative error of about 1.4% on mean code sequence power and approximately 0.10% on standard deviation; at lower sampling rates, these relative errors are slightly worse. *One caveat with aggressive sampling (such as 1%) is that we might only see fewer CSPPV samples, that may result in inability to accurately assess PNS, PS and PN probability values for certain basic blocks that are omitted due to sampling.*

C. Scalability of CSPPV Memory Log

We study the average and worst-case CSPPV memory footprint sizes for different numbers of cores after applying Lempel-Ziv compression. In other words, we measure the total memory needed for all of the CSPPVs from start to end of application execution. We note that the OS does not need to store the entire log, and could minimize the log size by periodically deallocating the memory pages that have already been processed by the software profiler. Figure 8 shows that the average-case memory requirements for many of our benchmarks are between 100 MB and 125 MB due to the efficiency of LZ compression (that offers up to 70% compression ratio). The worst case memory requirements are observed in fluidanimate benchmark which needs between 250 MB (4 cores) and 395 MB (32 cores).

D. Area, Power and Latency of Watts-inside Hardware

To obtain the area, power and latency of Watts-inside hardware, we create a Verilog-based RTL model of the power estimator, power analyzer, and hardware causation probability modules. We use Synopsys Design Compiler (ver G-2012.06) [20] and FreePDK 45nm standard cell library [19] to synthesize each module. Table VI-D shows the results of our experiments. We note that the area requirements for Watts-inside are modest and are about 0.2% of total onchip area of 4-core Intel Core i7 processor ($263mm^2$) [9]. Power requirements are less than 0.06% of 130 W peak power. *Since our hardware is designed to be off the critical path of the processor pipeline, we did not observe any significant performance impact in applications.*

TABLE II. AREA, POWER AND LATENCY ESTIMATES OF WATTS-INSIDE HARDWARE

	Power Estimator (×4)	Power Analyzer	Causation Prob. module	Buffer (4 KB)
Area(mm^2)	0.18	0.11	0.09	0.03
Power(mW)	9.08	8.72	4.53	49.70
$Latency^{\dagger}$ (CPU cycles)	24	$38^{\ddagger}$	$26^{\ddagger}$	NA^{*}

† Based on instruction latencies of Intel Core i7 [9]
‡ Not significant at runtime due to efficient filtering of code sequences
* Accessed when memory bus is free

E. Case study– Improving Load/Store Queue power consumption

In this subsection, we show how our Watts-inside framework offers a hardware-software cooperative solution in identifying and analyzing program code, and eventually improving the power consumption of the processor. Specifically, we pick two benchmark applications (ocean and streamcluster) that suffer from high load/store queue power.

We first modify SESC simulator to implement our framework. We then run our two case-study benchmarks (4-threaded version), and identify the portions of program code (shown in Figure 9) that consume high power. In ocean, the measured chip-level power is 73 W, and the instructions within the identified loop have $0.972 \leq PS \leq 1.0$, $0.70 \leq PNS \leq 0.72$. In streamcluster, the measured chip-wide power is 58 W, and the loop instructions under study have $0.968 \leq PS \leq 1.0$, $0.76 \leq PNS \leq 0.78$. In other words, for both benchmarks, Watts-inside indicate that the instructions within the loops have very high probabilities of sufficiency for high power within their corresponding code sequences.

```
//Ocean(Splash−2):main.cpp:337
...
for (i=numlev−2;i>=0;i−−) {
       imx[i] = ((imx[i+1] − 2) / 2) + 2;
       jmx[i] = ((jmx[i+1] − 2) / 2) + 2;
       lev_res[i] = lev_res[i+1] ∗ 2;
}
...
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

//Streamcluster(Parsec−1.0):streamcluster.cpp: 657
...
accumweight= (float∗)malloc(sizeof(float)∗points−>num);
accumweight[0] = points−>p[0].weight;
totalweight=0;
for( int i = 1; i < points−>num; i++ ) {
      accumweight[i] = accumweight[i−1]+points−>p[i].weight;
}
...
```

Fig. 9. Code snippets from ocean and streamcluster benchmarks where store-to-load dependencies result in high power

In both of the code sections, we observe a store-to-load dependency that results in a forwarding operation in load/store queue between the array elements across two iterations of the loop, i.e, the element that is stored in the previous iteration of the loop is loaded in the next iteration again. To reduce this unnecessary forwarding between the two iterations, we modify the code to include temporary variables that store the value from previous iteration and supply this value to the next iteration [1]. The code modifications are shown in Figure 10.

```
//Ocean(Splash−2):main.cpp: 337
//∗∗No more Store−to−Load Dependencies∗∗
...
t1 = imx[numlev−1];
t2 = jmx[numlev−1];
t3 = lev_res[numlev−1];
for (i=numlev−2;i>=0;i−−) {
       imx[i] = ((t1 − 2) / 2) + 2;
       jmx[i] = ((t2 − 2) / 2) + 2;
       lev_res[i] = t3 ∗ 2;
       //Storing array elements in temp
       t1 = imx[i];
       t2 = jmx[i];
       t3 = lev_res[i];
}
...
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

//Streamcluster(Parsec−1.0):streamcluster.cpp: 657
//∗∗No more Store−to−Load Dependencies∗∗
...
accumweight= (float∗)malloc(sizeof(float)∗points−>num);
accumweight[0] = points−>p[0].weight;
totalweight=0;
t = accumweight[0];
for( int i = 1; i < points−>num; i++ ) {
      accumweight[i] = t+points−>p[i].weight;
      //Storing array elements in temp
      t = accumweight[i];
}
...
```

Fig. 10. Modified code snippets from ocean and streamcluster benchmarks that no longer have store-to-load dependencies

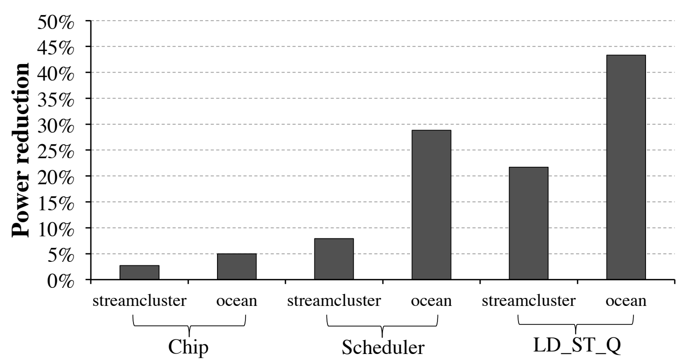

Fig. 11. Power reduction across the entire chip, scheduler and load/store queue after removing store-to-load dependencies

As a result of this code optimization, we find an improvement in chip-wide power consumption in both benchmarks. An interesting side-effect of our code modification was the reduction in the number of memory load instructions in each loop iteration due to replacement of memory load with operations on temporary registers, that consequently showed reduction in scheduler power. Figure 11 shows the results of our experiments. In streamcluster, we observe an average savings of 2.72% for chip power (and up to 21% reduction in load/store queue and 7% savings in scheduler power consumption) with a slight 0.25% speedup in execution time; In ocean, we get an average savings of 4.96% in chip power (and up to 43% reduction in load/store queue power and 28% savings in scheduler power consumption) with a slight 0.19% speedup in execution time.

From this case study, we observe the usefulness of understanding application power and how the feedback information can be utilized in meaningful ways to improve power behavior of multicore applications. We note that, in this particular case study of removing store-to-load dependencies, many compilers typically are unable to optimize code in a way that avoids store-to-load-dependency [1]. In some cases, the language definition prohibits the compiler from using code transformations that might remove store-to-load dependency. Therefore, a framework like Watts-inside, that offers a hardware-software cooperative solution to understanding and improving multicore power, can be an invaluable tool for multicore software developers.

VII. Related Work

Prior works [3], [10] have considered using performance counters for processor power modeling. CAMP [15] and early stage power modeling [11] showed how to use limited number of hardware statistics to estimate power consumption by microarchitecture units. Our Watts-inside hardware can leverage these techniques to estimate the power drawn by different processor components.

Tiwari et al. [21] developed an instruction-level power model that attributes energy to program instructions. Powerscope [7] utilizes hardware program counters to provide procedure-level feedback. Such software-only methods generally find it difficult to accurately estimate microarchitecture power due to complex interactions between functional units. Also, the performance overheads of such tools and simulators can be high, making them hard to use in production environments. In contrast, our Watts-inside proposes a hardware-software cooperative solution, where hardware provides fairly accurate information about program execution and the software offers flexible platform to analyze program power.

VIII. Conclusions and Future Work

In this paper, we showed the necessity to gather fine-grain information about program code to better characterize application power and effect improvements. We proposed Watts-inside, a hardware-software cooperative framework that relies on the efficiency of hardware support to accurately gather application power profiles, and utilizes software support and causation principles for a more comprehensive understanding of application power. We presented a case study using two real applications, namely ocean (Splash-2) and streamcluster (Parsec-1.0) where, with the help of feedback from Watts-inside, we performed relatively straightforward code changes and realized up to 5% reduction in chip power and slight improvement ($\leq 0.25\%$) in execution time.

As future work, we will extend our framework to incorporate uncore processor structures such as network, tag-directories, and memory controllers. We also plan to include more challenging computing environments such as heterogeneous processors.

References

[1] "Software optimization guide for the amd64 processors," http://support.amd.com/us/Processor_TechDocs/25112.PDF.

[2] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Princeton University Technical Report TR-811-08*, January 2008.

[3] W. Bircher and L. John, "Complete system power estimation: A trickle-down approach based on performance events," in *ISPASS*, 2007.

[4] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of ISCA*, 2000.

[5] J. Chen, G. Venkataramani, and G. Parmer, "The need for power debugging in the multi-core environment," *IEEE Computer Architecture Letters*, 2012.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of ISCA*, 2011.

[7] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proceedings of WMCSA*, 1999.

[8] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the adaptive energy management features of the power7 chip," *Micro, IEEE*, 2011.

[9] Intel Corporation, "Intel core i7-920 processor," *http://ark.intel.com/Product.aspx?id=37147*, 2010.

[10] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *MICRO*, 2003.

[11] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer, "Abstraction and microarchitecture scaling in early-stage power modeling," in *Proceedings of HPCA*, 2011.

[12] N. P. Jouppi *et al.*, "Cacti 5.1," *http://quid.hpl.hp.com:9081/cacti/*, 2008.

[13] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[14] J. Pearl, *Causality: models, reasoning and inference*. Cambridge Univ Press, 2000, vol. 29.

[15] M. D. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, and S. Yardi, "Camp: A technique to estimate per-structure power at run-time using a few simple parameters," in *Proceedings of HPCA*, 2009.

[16] Process Integration, Devices and Structures, "International Technology Roadmap for Semiconductors," *http://www.itrs.net*, 2007.

[17] J. Renau *et al.*, "SESC," *http://sesc.sourceforge.net*, 2006.

[18] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, 2012.

[19] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *Proceedings of MSE*, ser. MSE '07, 2007.

[20] Synopsys Inc., "Design Compiler User Guide," *http://www.synopsys.com*, 2012.

[21] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Process. Syst.*, vol. 13, no. 2-3, Aug. 1996.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Proceedings of ISCA*, 1995.

[23] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theor.*, vol. 23, no. 3, Sep. 2006.