Live, Runtime Power Measurements as a Foundation for Evaluating Power/Performance Tradeoffs

Russ Joseph, David Brooks, and Margaret Martonosi
Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544-5263
{rjoseph, dbrooks, mrm}@ee.princeton.edu

ABSTRACT

Of the many ways one could gauge the complexity-effectiveness of a design or design element, one candidate approach is to consider a design's power/performance tradeoffs. This paper describes our early-stage results in a broad effort to evaluate the power-performance tradeoffs of a range of benchmarks and microarchitectures. In particular, this paper presents power data collected on-the-fly on real x86 machines as they execute carefully-constructed microbenchmarks. The microbenchmarks exercise aspects of the system such as data cache and branch predictor. They are parametrically-variable to consider how load dependence, cache miss rate, branch mispredict rate, and branch distance all impact the power and performance of a CPU. For example, from these experiments, we learn that CPU performance increases essentially monotonically with cache hit rate, while CPU power encounters a maximum at roughly 80-90% cache hit rates. Likewise, we show results demonstrating that performance-neutral issues such as bit populations in the data cache values can display interesting power trends. While the experimental results are preliminary, we feel that the techniques described in this paper will offer a useful foundation for a broad range of power/performance tradeoffs.

1. INTRODUCTION

The notion of complexity-effective design can mean different things to different designers. At its most basic level, however, it means that the design elements should give benefits that are commensurate with their cost. Benefits of a design choice can include factors like higher performance or increased reliability. Metrics for evaluating a design choice's cost might include transistor counts, CPU yield rates or power dissipation. The exact preferences for benefits and cost depend on the type of the CPU being built and its intended usage.

In this paper, we focus on comparing performance benefits to power costs for particular design elements. The platform for these experiments involves real CPU power and performance measurements developed for our Castle power estimation project [6]. Our Castle work gives accurate power profiles for programs as they execute. An exploration of the interplay between power and performance effects presented in this paper seemed a natural extension.

In order to isolate particular performance effects, we have developed a benchmark generator we call TraumaGen. As the name implies, TraumaGen allows one to set up assembly language codes with particularly diabolical—and parametrically controlled— behaviors. That is, by inputting machine characteristics, we can get from TraumaGen a benchmark with a given instruction or data cache miss rate, or one with a particular branch mispredict behavior.

The synthetic microbenchmarks created by TraumaGen can be put to many uses, but in this paper we use them for comparing the power costs and performance benefits of a particular design. This allows us to get one sense of the complexity-effectiveness of the design choices. The work described in this paper is a start on a much larger effort to characterize power/performance tradeoffs using real-machine measurements on a range of programs and CPUs. We present some initial results here. While the early results are already of interest on their own, the paper's description of our technique is, we feel, also a contribution since it is general and promises useful future data. The design we focus on here is the Pentium Pro microarchitecture.[1] While Pentium Pro seems somewhat dated, the core microarchitecture of the Pentium Pro is similar to PII and PIII as well. The next step in our research will be to broaden our approach to other newer architectures.

The remainder of this paper is structured as follows. Section 2 gives an overview of the power and performance measurement strategy used. Section 3 describes our TraumaGen microbenchmark generators. Section 4 outlines the methodology used in this paper. Sections 5, 6, and 7 then present power/performance tradeoff studies on data caches, branch predictors, and bit activity levels, respectively. Finally, Section 8 offers a summary and discussion of future work.

2. POWER AND PERFORMANCE MEASUREMENT

For our study, we need both accurate power and performance data. Fortunately, we had previously developed a scheme for observing and recording processor power as part of our Castle power estimation project [6]. Castle gathers usage statistics from performance counters and can determine processor component and total power consumption as a program executes. Here we present the power measurement

[1] We will admit that it was chosen because as the least valuable machine in the lab, it seemed the best candidate for being opened up and having its power lines cut and manipulated!

approach, and briefly describe our approach for producing performance statistics.

2.1 Power Measurement Using Castle

Castle power estimation uses performance counters and data sampling to produce processor power profiles, including component breakdowns. As part of our Castle development and verification, we also devised a scheme for direct CPU power measurement. Figures 1 and 2 give examples of the power windows displayed with live Castle power readings as the Olden benchmark *health* [10] is executed.

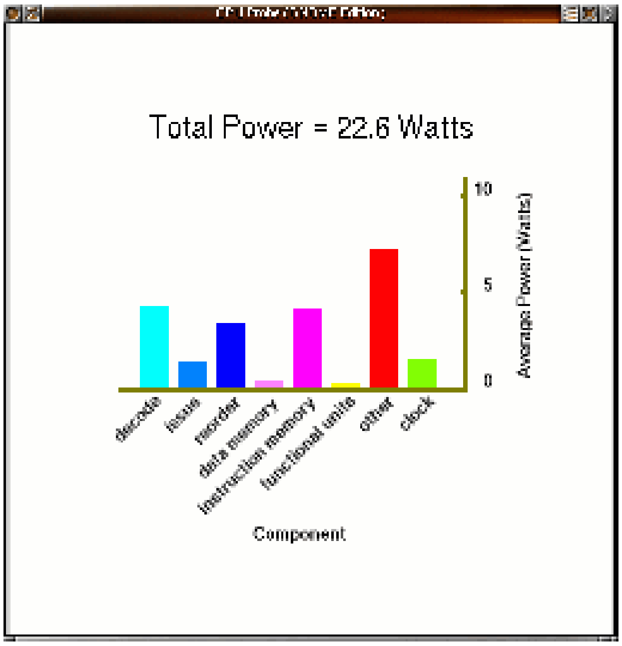

Figure 1: Performance counter based Castle power profile for *health*

Figure 2: Direct power measurement for *health*

For this work, we focus on the total power readings given by direct measurement. While the power breakdowns will likely prove useful in future work, we do not use them for results given in this paper.

To collect processor power statistics we apply the scheme illustrated in Figure 3. This is similar to the approach discussed for Compaq Itsy power measurements [13]. First, we cut the connection between the internal power supply, and motherboard. We then placed a shunt resistor in series

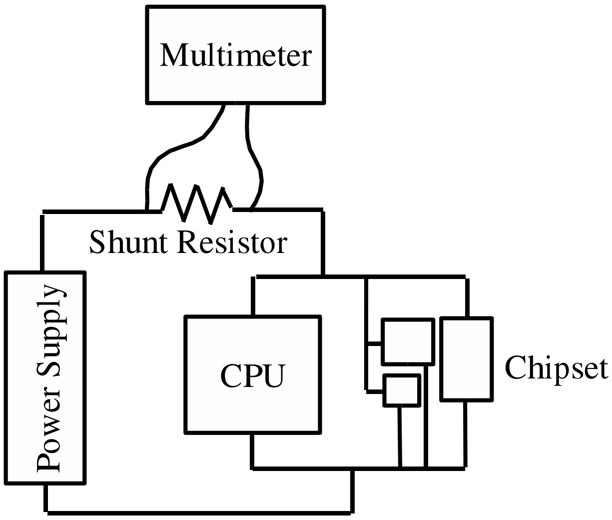

Figure 3: Diagram of Power Measurement Setup

with the supply and motherboard. At any point, the current through the shunt resistor is identical to the current through the CPU and chipset.

Figure 4: Photograph of Measurement Setup

To deduce the processor and chipset power, we relied on P = V * I. The power supply line under examination was known to deliver a constant 5V. To determine the current through the shunt resistor, we attached a HP 34401A Digital Multimeter to both terminals of the shunt resistor. Since the resistance was known, we were able to calculate the current through the resistor and hence through components considered with Ohm's Law (V = I * R). This immediately gave us the power to the CPU and chipset. Figure 4 shows a photograph of the system being measured. The multimeter pictured is connected via a serial cable to a second PC that collects data over time.

To separate processor from chipset power, we measured power consumption of the processor in idle mode and compared this to published information about the Pentium Pro CPU's idle mode power [5]. The difference between our measured value and the value published for the CPU alone is assumed to be the chipset power. We assume that the chipset power is constant in our benchmarks and that only the CPU power varies, since we do not expect chipset elements like bus controllers to vary in power extensively during these microbenchmarks. Since our benchmarks are constructed to minimize off-chip accesses after start up, we feel this is reasonable. (We hope to refine these assumptions further in future work.)

For the results presented here, our experimental platform was an Intel Pentium Pro 200MHz Linux workstation with 128 MB RAM and a 2 GB hard drive. The system was running the Red Hat Linux 2.2.16-3 kernel. While the system is fairly dated, it served as a good initial platform for our studies because it has a complex enough CPU to be interesting, and yet is widespread enough that there are good specification documents widely available on the power lines to the CPU and motherboard. Furthermore, more recent chips like the Pentium II and Pentium III have a nearly identical microarchitecture to the Pentium Pro, although they run at higher clock rates. We should expect many of the trends identified in this study to hold for other members of the Intel P6 family, although the magnitude of the tradeoffs will vary.

2.2 Performance Measurements

In order to perform detailed performance/power tradeoff studies, we need extensive performance information about the programs being run. Towards that end we employed the CPU's hardware performance counters [9]. These performance debugging aids are nearly ubiquitous in modern microprocessors. They can typically be used to tabulate important processor level events like cache hits/misses, branch mispredictions, or instruction retirement. In our case, the specific counters that were the most useful were: (i) execution cycle counts, (ii) instructions decoded, (iii) instructions retired, (iv) branch instructions decoded, (v) branch instructions retired, (vi) branch mispredictions, (vii) memory references, (viii) L2 cache accesses (as an indicator of L1 cache misses).

While our microbenchmarks were designed to explore the effect of various microarchitectural events, we also needed to independently verify that the desired occurrence rates were actually being sustained. For example, when we attempt to create a suite of microbenchmarks with L1 cache miss rates as close as possible to 10%, 20%, 30% and so on for our tradeoff graphs, we need to know how close we got. We also need to check that the IPC achieved verifies that we created assembly code with the expected number of dependent instructions on each load. While these checks can clearly be gotten by visually examining the microbenchmark assembly code, we use performance counters as an additional sanity check. In particular, to calibrate the accuracy of our microbenchmarks, we measured critical architectural events with the Rabbit PMC Library [4]. This toolset affords developers easy access to the Pentium Pro's performance counters. All performance-relevant statistics presented in this work (such as the IPC and DPC graphs in Figures 9 and 13) were gathered from hardware performance counters with this library. (While Castle also uses hardware performance counters, it relies on kernel modifications to read the particular counters of interest. Our performance studies need more flexibility in which counters are read and when; for this reason, Rabbit proved useful here.)

3. THE TRAUMAGEN MICROBENCHMARK GENERATOR TOOLSET

In order to exercise the measured system in a known way, we need to write microbenchmarks that have particular well-constrained behaviors [11]. For example, Section 5 needs microbenchmarks with carefully controlled reference rates and data cache miss rates, while Section 6 studies different branch rates and mispredict frequencies. To help us in generating such codes, we first developed the TraumaGen microbenchmark generator toolset. Each TraumaGen generator is a C language program, intended to produce a different type of microbenchmark. These include exercisers of the data and instruction caches, the branch predictors, as well as bitline activity exercisers.

As input, each TraumaGen generator takes parameters about the machine being studied (such as the cache size and organization). As output, it produces an assembly language program with the desired characteristics. We plan to build TraumaGen generators for a variety of RISC and CISC architectures, but this paper uses ones for building microbenchmarks in the x86 instruction set architecture.

3.1 TraumaGen Example

To better explain TraumaGen's capabilities, we present here a running example of generating data cache exercisers using TraumaGen. In this case, TraumaGen takes as inputs the desired cache hit rate and dependence information. Using these plus a knowledge of the target microarchitecture and instruction set architecture, the generator produces a microbenchmark with these requested characteristics.

Figure 5 provides a generic overview of the structure of the microbenchmark TraumaGen produces for this cache exercise example. TraumaGen can generate load instructions to guarantee that a cache hit or miss will occur.

```
TOP:
   LOAD CACHE_MISS
   ADD                  Possible
   ADD                  Dependencies
   LOAD CACHE_HIT
   ADD
   ADD
          .
          .
          .
   LOAD CACHE_HIT
   ADD
   ADD
BRANCH TOP
```

Figure 5: Structure of L1 D-Cache microbenchmark as generated by TraumaGen

Linked list access which results in cache miss

Memory access which results in cache hit

```
movl (%eax), %eax
addl $1967513926, %ecx
addl %eax, %edx
movl (%ebx), %edi
addl $1303455736, %ecx
addl %edi, %edx
movl (%eax), %eax
addl $1540383426, %ecx
addl %eax, %edx
movl (%ebx), %esi
addl $521595368, %ecx
addl %esi, %edx
```

Instruction which is independent of preceding load

Instruction which is dependent on preceding load

Figure 6: Assembly code produced by TraumaGen for the case of a microbenchmark with a 50% L1 D-Cache miss rate.

Figure 6 shows a specific example of the assembly code TraumaGen produces for the case where we desire:

- 50% L1 Cache miss rate
- 1 dependent instruction and 1 independent instruction per memory reference
- 8KB L1 data cache with 32 byte lines (2-way set associative)

For this case TraumaGen produces code with a long loop of roughly 800 instructions. The key basic block is intentionally large to minimize the effects of branch instructions and branch predictions. On the other hand, the basic block is sized to be small enough to reside wholly in the L1 instruction cache, to minimize the effects of Icache misses. The loop does repeated load instructions to addresses chosen to interfere in the L1 data cache at a particular rate. Since the total number of pages referenced by the program is quite small, TLB effects are not an issue; we applied several sanity checks to verify this. Since “real” programs do not simply bombard the memory system with streams of solely memory instructions, our microbenchmark intersperses ALU instructions between the loads. The Pentium Pro used in this study can issue up to three instructions per cycle, so our benchmarks feature a single memory reference and two ALU instructions per cycle. Finally, we chose to explore some of the power/performance effects related to memory dependent instructions, so we vary the dependence of the ALU instructions on the preceding load. In Figure 6, the user requested 1 dependent instruction and 1 independent instruction per memory reference.

While this example focuses on TraumaGen’s use for memory system microbenchmarks, we have also applied similar techniques to generate benchmarks that stress the branch prediction hardware and that exercise bitlines with given activity factors. Such approaches are discussed in more detail in Sections 6 and 7.

4. EXPERIMENTAL SETUP

While the Intel Pentium class systems are fixtures of the desktop computing landscape, many are not immediately familiar with the microarchitecture that drives them. Here we supply a brief introduction to the P6 Microarchitecture and follow with our procedure for generating microbenchmarks and analyzing the performance/power effects witnessed by them.

4.1 Pentium Pro Microarchitecture

The P6 Microarchitecture is the underlying implementation that fuels most of Intel’s family of processors including the Pentium Pro (used in this study), Pentium II, and Pentium III. A common feature is the ability to convert variable length CISC instructions into simple, fixed size micro-operations, which are fed to the out-of-order execution engine [3]. This is all done while maintaining fairly high performance.

To support its deep superscalar pipeline, the Pentium Pro processor relies on very clever instruction fetch and decode logic. While the details of the decode engine are not the focus of our study, this logic does draw a considerable amount of power, and entails significant complexity[8, 3]. The fetch logic, particularly, the branch prediction implementation are explored in our study, so we will examine some of its salient details.

The highlight of P6 branch prediction is the 128 set, 4-way associative branch target buffer (BTB) [7]. In addition to tag and status bits, each BTB entry records branch target, and speculative and non-speculative versions of a local 4-bit branch history register. The branch history registers are used to record the taken/not-taken behavior of static branches. The four entries of a BTB set all share a pattern history table. In this manner, the P6 supports two-level adaptive branch prediction [14]. Our work examines some of the power related effects of branch prediction.

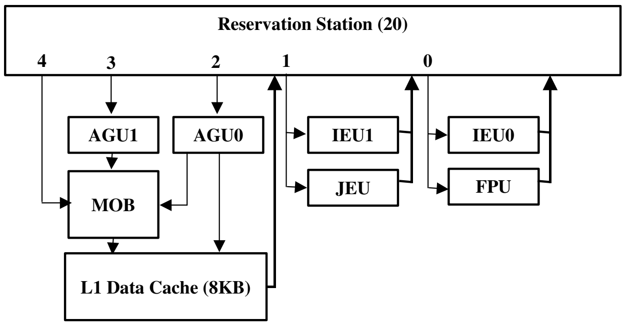

Figure 7: P6 Execution Engine

The P6 out-of-order engine pictured in Figure 7 features five functional units and supports issue of three translated RISC micro-ops [3]. It features a 20 slot centralized reservation station which allows a modest amount of out-of-order execution. One complex functional unit pipe supports all basic integer operations, as well as multiplication, division and floating point operations. A second functional unit pipe only handles basic integer and branch instructions. Most operations are fully pipelined. The memory units feature a store buffer with forwarding to accompany the standard 8 KB cache. Loads typically execute in three cycles.

4.2 Experimental Procedure

For each study, we constructed a separate TraumaGen microbenchmark generator. The microbenchmark generators were then employed to produce the executables used in this study. The benchmarks were tailored to examine the desired features. Care was taken to limit unwanted architectural events (e.g. unwanted cache misses, or I/O access).

We measured the average power consumed by each benchmark with the approach described in Section 2 and pictured in Figure 4. The multimeter data logger was configured to produce 50 sample readings per second. Typical sample variance for a given benchmark was on the order of 0.05 W. Each individual benchmark was run for at least 5 minutes. This corresponds to 60 billion cycles of observation and a minimum of 18 billion instructions.

After measuring the power consumption of these benchmarks, we used the Pentium Pro hardware counters to monitor the performance characteristics of the benchmarks as described in Section 2.

Finally, with both the power and performance statistics for our study, we analyzed the available data and Sections 5, 6, and 7 present a detailed discussion of the trends.

5. DATA CACHE

The first level data cache has a critical impact on both the performance and power dissipation of most modern microprocessors including the Pentium Pro. We have studied the performance and power dissipation of the L1 data cache to gauge the complexity-effectiveness of this part of the microarchitecture. To perform this study, we use TraumaGen as described in Section 3 to generate cache microbenchmarks that expose the power and performance effects of cache miss rates and load value dependencies. In this section, we use TraumaGen microbenchmarks with the following characteristics:

- All load hits in the benchmark access the same cache line. All memory accesses are controlled, so this line is never replaced and always results in a hit.
- All L1 load misses are performed in a sequential manner to a group of conflicting cache lines determined by cache associativity and sizing parameters. The memory accesses are chosen so that they will always access replaced lines.
- No load instructions generate L2 misses. The blocks generating L1 misses are chosen so that they always yield L2 hits. This is crucial since we want to minimize the contributions of off-chip access.

These memory referencing instructions (depicted as load_cache_miss and load_cache_hit in Figure 5) are then generated in random order to meet the ratio specified by the user. Since the Pentium Pro can issue two integer operations in addition to a load operation, two addition instructions are also generated along with each load operation. The user can specify the amount of dependence that these addition operations will have on their corresponding load operation; no dependence, one operation dependent, or both operations dependent. A large loop operation ensures that the microbenchmark will run long enough to allow power and performance measurements to be accurate.

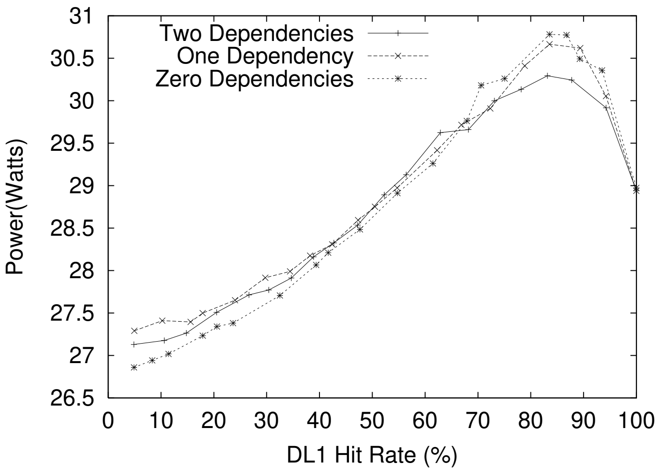

Figure 8: Power Consumption vs. Cache Hit Rate

Using TraumaGen and our performance and power measurement infrastructure, we have measured the effect of cache misses and dependencies on power consumption and IPC. Figure 8 shows power consumption on the Y-axis and L1-Dcache hit rate on the X-axis. Three sets of data are plotted on this graph. The first set of data is the case where there are no dependencies between the load operation and the two addition operations that are decoded on each cycle. The second and third sets of data show the results when 1 and both of the addition operations are dependent on the result of the load operation.

Several interesting trends are revealed by this data. First, power dissipation peaks at a cache hit rate of approximately 80-85%. Two conflicting actions are occurring within the microarchitecture to cause this phenomenon. With very good cache behavior, the nonblocking cache miss support, L1-L2 interconnection network, and L2 cache will not be exercised frequently resulting in lower power dissipation. On the other hand with very poor cache behavior, the main pipeline will be stalled more frequently because of dependencies and the saturation of the outstanding miss support hardware. Stalls, or bubbles, in the main pipeline will drastically reduce power dissipation.

The second interesting trend is related to the amount of dependencies in the microbenchmark. At the peak in power around 85% hit rate, the 1- and 2-dependency microbenchmarks have lower power dissipation than the 0-dependency microbenchmark. The reason for this is because as the number of dependencies increase, there will be more stall cycles in the main pipeline waiting for these L1-cache misses to be resolved. As the cache hit rate decreases below 50%, this trend reverses; at this point, the 0-dependency microbenchmark uses less power than the 1- and 2-dependency microbenchmarks. A possible reason for this is because additional forwarding and rename logic is being exercised to provide dependent operations with their data values.

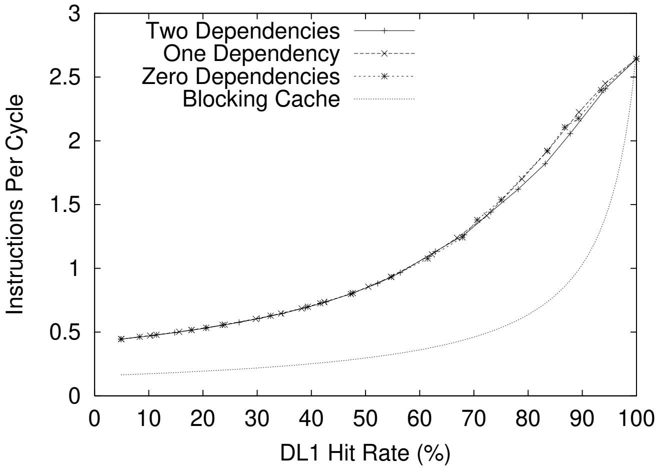

Figure 9: IPC vs. Cache Hit Rate

Figure 9 shows the IPC measurements for the same suite of microbenchmarks (read with our hardware performance counter setup) and the IPC for a microarchitecture with a blocking cache. The IPC for the blocking cache was computed with the following formula:

$$IPC = \frac{1}{(CPI_{Base} + PProL2Penalty * (1 - HitRate))} \quad (1)$$

The IPC drops for all three sets of microbenchmarks from a base IPC of 2.7 instructions per cycle to around 0.5. (The base IPC reflects the case where there are zero dependent instructions per load, and the cache is ideal.) However as witnessed in Figure 9, the non-blocking caches on the Pentium Pro give a significant performance increase over the blocking cache. Given this significant performance benefit, the additional power dissipation suggested in Figure 8 is complexity-effective when power is used as the metric of complexity.

6. BRANCH PREDICTION

While the impact of branch prediction on superscalar performance has been well documented, its effects on power are not well-understood. To support large degrees of speculation and high branch prediction rates, high performance processors employ very sophisticated branch prediction schemes that add considerable complexity [15]. To examine some aspects in the complexity-effectiveness of the Pentium Pro branch prediction mechanism, we designed an experiment again using our TraumaGen toolset.

In many processors, including the Pentium Pro, unconditional and conditional branches have different mispredict penalties. This follows since unconditional branches can be discovered during decode, while mispredicted conditional branches are not identified until execution. We focused our study on conditional branches. In particular, we explored the effect of accuracy of branch direction prediction versus the frequency of branches. We took care to ensure that our microbenchmarks were small to minimize cache and BTB misses. However, our sample space had to be large enough to give accurate results, and minimize loop effects (e.g. easily predicted backward loop branches).

We only examined branch prediction accuracies greater than 50%. This seemed reasonable since most programs have prediction accuracy rates above 50%. In addition, we also increased the average number of cycles between branches from 1 to 10. While it is quite possible for a wide superscalar machine to see two or more branches per cycle, this would typically happen infrequently.

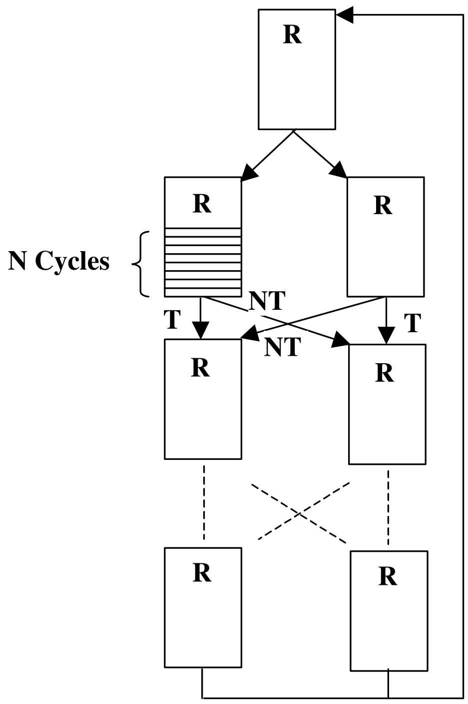

Figure 10: Structure of Branch Prediction Experiments.

Figure 10 illustrates our branch experiments. For every conditional branch in our example, the code appearing at the branch fall through is identical to the code at the target. This was done to ensure that the power-wise contribution of either arm of the branch would be equivalent. To achieve specified branch prediction accuracy, we replaced the code segments including random branches (denoted with 'R's) with new branches that will be predicted correctly after a short warm-up. This allowed us to target arbitrary accuracies in the 50-100% range. The other variable examined in our experiments is the spacing between branches in cycles. Figure 10 shows the flow diagram when the distance in cycles between branches is N. The cycles between branches are filled with simple, high IPC code that accesses the data cache and function units. Figure 11 shows a snippet of microbenchmark code where there are two cycles between branches.

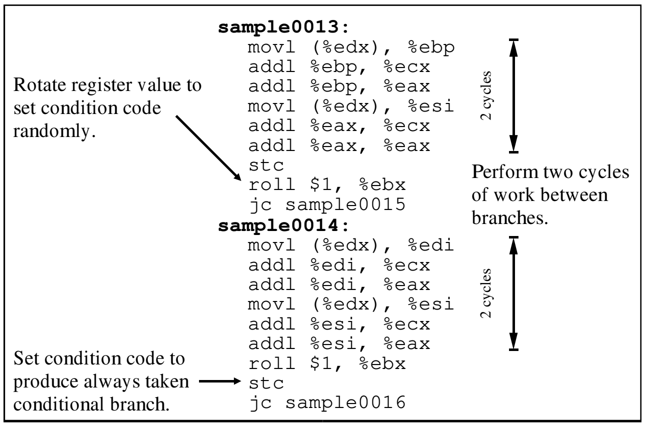

Figure 11: Assembly code produced by TraumaGen for the case of a microbenchmark with 2 cycle spacing between branches.

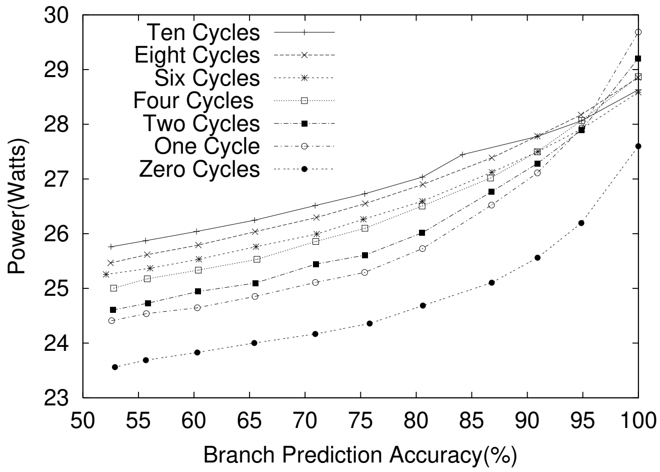

Figure 12: Power vs. Branch Prediction Accuracy.

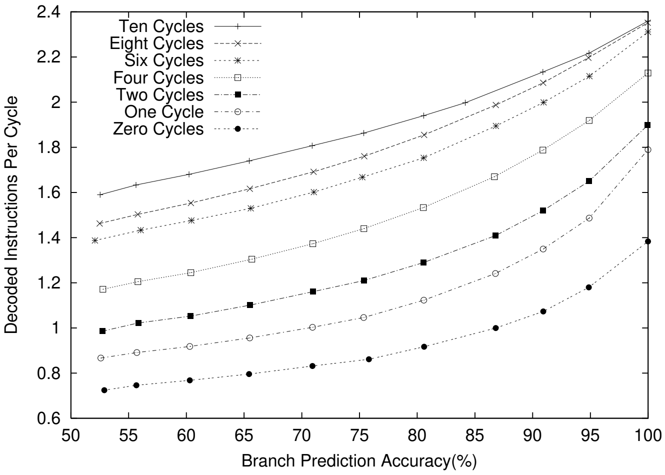

Figure 13: Decoded Instructions per Cycle vs. Branch Prediction Accuracy.

While it could be difficult to cause a given BTB entry to always make an incorrect prediction, it was easy to make branch conditions very erratic. It was also easy to make certain branch conditions extremely predictable. To produce desired prediction accuracies, we varied the ratio of predictable and unpredictable branches. Our general approach is outlined below.

- To produce erratic branches, we made the branch outcome dependent on data dependent bit rotations of a specific register. The number of zeros and ones in the data register is kept equal, so the branch is equally likely to be taken or not-taken, but the sign of the value changes in a bizarre pattern that ordinary branch predictors cannot comprehend.
- For each predictable branch, the outcome was made dependent on a condition flag that was either always set or always clear.
- Finally, for the easily predicted branches, we chose to equally distribute taken/not-taken outcomes.

Figure 12 shows the power for given prediction accuracies when the distance between branches varies from 1 to 10. Figure 13 shows the number of instruction decodes per cycle (DPC). We chose this metric rather than IPC because it better reflects the notion of total work versus necessary work that is key to understanding energy related effects of branch prediction [8]. We will use the DPC presented in Figure 13 as tool for examining power relevant features in Figure 12.

Several interesting trends appear in Figure 12. In general, power increases as branch prediction accuracy improves for all basic block sizes. This is not surprising since an increase in prediction accuracy means that the pipeline has to flush fewer instructions. Since stalls are essentially nops that can be clock gated, they draw less instantaneous power than executing real instructions. This explanation is further supported by noting that DPC also increases with accuracy.

When there is no extra work between branches, the power consumption drops off considerably. This is not surprising since there are no additional uses of functional units, data cache, or any other resources. Non-contiguous instruction fetches also introduce some pipeline bubbles, which account for even lower utilization. Finally, with branches being executed this frequently, speculation depth is most likely reached very quickly, promoting further stalls and limiting power usage. So, even when the prediction rate is very high, power consumption remains low.

In most cases, an increased distance between branches translates into larger power consumption. This is intuitive since a larger portion of the code contains compute instructions which are obviously higher power than pipeline bubbles. As basic block size and the number of compute instructions increase, the relative impact of pipeline bubbles introduced by mispredictions diminishes.

Perhaps the most curious feature of Figure 12 is that when the distance between branches is a small non-zero number of cycles, the power consumption overtakes larger branch distances as the accuracy approaches perfect. In particular, consider a branch cycle distance of one. The processor power for this case exceeds all other branch cycle distances by a large margin at 100% accuracy. However, Figure 13 shows that DPC always increases as distance between branches increases, regardless of prediction accuracy.

This effect can be explained as follows. When the number of extra cycles is small, but larger than zero, functional units and the rest of the execution hardware are utilized enough to maintain moderate power levels when prediction accuracy is good. As prediction accuracy approaches the 95-100% range, the execution units are being kept busy, pipeline bubbles are rare, and the branch hardware is significantly utilized. This accounts for the increased power consumption. When the distances between branches are larger, the branch units are not utilized enough to witness large power increases. This explains why the DPC numbers may be higher without a corresponding jump in power.

7. DATA ACTIVITY

In several recent studies, architectural techniques have been proposed to save energy by reducing the amount of unnecessary switching within the processor [1, 2, 12]. Reducing data switching activity when possible is an attractive means to reduce power dissipation because in some cases significant savings can be achieved with no performance impact. In this section, we seek to gauge the potential for these types of techniques by exploring the effect of data activity factor on power dissipation within the L1 Data Cache of the Pentium Pro.

One way to gauge the effect of data activity factors is to look at the measured power consumption of the Pentium Pro as we read various values out of the data cache. The microbenchmarks to perform this is fairly trivial. A data value is passed as a parameter to TraumaGen, and a microbenchmark is created which repeatedly reads this value out of a fixed memory location. By changing only the data value in the microbenchmarks, we can isolate the effect of data activity factor.

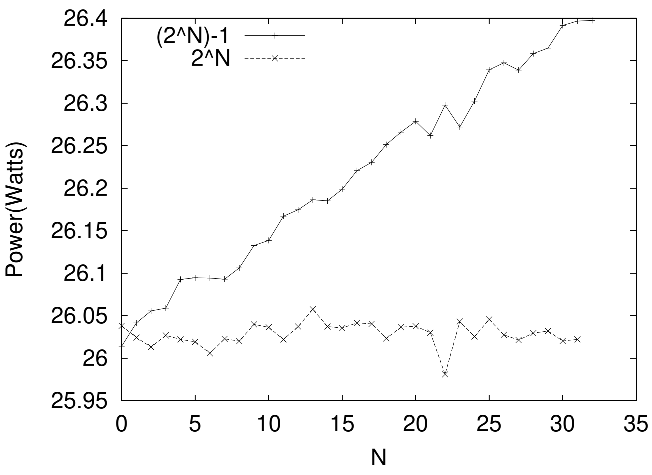

Figure 14: Effect of Data Activity Factor within the DCache on Power Dissipation.

Figure 14 shows power dissipation with various amounts of data activity within the L1 Data Cache. To quantify the effect of data activity on power dissipation, we have considered reading values of 2^n and $2^n - 1$ while varying n from 0 to 32. These values correspond to the case when the nth-bit is set and when all of the first n-bits are set in the data value being read from the cache. Figure 14 shows that the power dissipation is relatively constant as we vary the values with 2^n. On the other hand, there is a clear linear increase in power dissipation when varying the values from $2^n - 1$. This trend means that something in the cache's path to the registers consumes more power for 1's than for 0's. It could either be a pre-charged bus implementation, single-ended bitlines in the data cache, or some other similar effect. We feel it is most likely that the cache structure is implemented with single-ended bitlines where values of 1 cause the bitline to evaluate; as we increase the number of 1s in the value being read from the cache, the number of bitlines pre-charging and evaluating on each cycle increases, causing the power dissipation to increase. If double-ended bitline structures had been used in this structure, which are also common in caches, identifying and capitalizing on cases to reduce the data activity factor may be fruitful.

From Figure 14 it is clear that data activity factor can have an appreciable effect on power dissipation within cache structures. The measured range in power dissipation was around 0.4W; from the Pentium Pro power breakdown data presented in [8] we estimate that the worst case power dissipation of the Data cache unit is slightly over 2W. While our current experiments do not let us draw conclusions about the effect of data activity factor within other structures, we do feel that it can play a significant factor as a performance-invariant index of power dissipation.

Although a 0.4W difference seems to be a small fraction of overall system power, our studies were restricted to a portion of the datapath. We are planning an extensive analysis of data activity in the rest of the processor, which may uncover more a drastic influence of data activity on total power.

8. SUMMARY

This paper represents a first-step in a broader effort to characterize power/performance tradeoffs across a range of architectures. Live power measurements allow us to avoid some of the inaccuracies of simulation based approaches. The TraumaGen microbenchmark generator toolset allowed us to create microbenchmarks with particular cache, bit activity, and branch prediction behaviors. With these microbenchmarks, we compared power and performance responses to a range of situations. The immediate data is interesting because it represents some of the most detailed live data published on superscalar, out-of-order processors. We expect to broaden the approach and the results to encompass more CPU design elements and a wider range of processors.

9. REFERENCES

[1] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.

[2] R. Canal, A. Gonzalez, and J. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec. 2000.

[3] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, February 1995.

[4] D. Heller. Rabbit: A Performance Counters Library for Intel/AMD Processors and Linux. http://www.scl.ameslab.gov/Projects/Rabbit/index.html.

[5] Intel Corp. Intel Quickstart Technology. http://www.intel.com/mobile/technology/management.htm.

[6] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, Aug. 2001.

[7] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, Jan. 1984.

[8] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 132–41, June 1998.

[9] T. Mathisen. Pentium secrets. *Byte Magazine*, pages 191–192, July 1994.

[10] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. In *ACM Transactions on Programming Languages and Systems, 17(2)*, March 1995.

[11] R. Saavedra, R. Gaines, and M. Carlton. Micro benchmark analysis of the KSR1. In *Proceedings of Supercomputing '93*, Nov. 1993.

[12] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec. 2000.

[13] M. A. Viredaz and D. A. Wallach. Power evaulation of itsy version 2.3. Tech. Note TN 57, Digital Western Research Laboratory, October 2000.

[14] T. Yeh and Y. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, November 1991.

[15] T. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, 1993.