

# Evaluating Instruction Reorderings and Transformations for Microarchitecture Power Reduction

**Ram Srinivasan**      **Jeanine Cook**  
New Mexico State University  
(ram,jcook)@nmsu.edu

IBM Technical Contact: **Lee Eisen**  
STSM, eClipz Development

## 1. Introduction

The strong emergence of battery operated, portable devices has made power consumption one of the most critical design constraints in modern processors. Apart from battery life, maintaining the processor at operable temperatures is becoming a cause for concern as we approach the limits of air-cooling. Considerable research has been done on trying to minimize processor power consumption through hardware techniques such as clock gating and voltage/frequency scaling and software techniques that try to minimize circuit switching. The primary goal of our research is to identify software optimization techniques for minimizing microarchitectural power consumption.

Our initial idea was to minimize power through activity factor reduction. For this we wanted to analyze the instructions within basic blocks and identify sequences that consume high power. Transforming such sequences through instruction re-ordering and substitution will minimize the amount of switching, resulting in a power reduction. Instruction substitution is a process wherein, a high power instruction is replaced by one or more low power instructions that are semantically equivalent. For substitution, the power/energy consumption data for each instruction of the ISA is necessary. The lack of such data for the POWER architecture, prompted us to work on this first. Though accurate power measurements can be achieved through direct-measurement on real hardware, we used the Turandot/PowerTimer simulator for our study due to resource constraints. Over the course of our study, it was determined that the simulator does not model activity factors but, instead relies on unit utilization for its power model. This made the simulator unsuit-

able for our study. We were also simultaneously working on the alpha architecture, for which we had the resources to directly measure power. While measuring per-instruction power using synthetic executables, we noticed that the processor power almost always decreases from its steady state value. We determined this was due to the high power operations performed by the OS in idle state. We modified the kernel and this resulted in a 6.75W reduction in idle state power with no visible performance impact. It was also determined that instruction (iword) induced activity factor has very little effect on the overall processor power consumption. Therefore, we have decided to refocus our research to study speculation control and memory behavior improvements through better compilation of the code. The knowledge that we have gained from our earlier study will help us in achieving this goal. This paper briefly describes the work done thus far and outlines our plans for future.

First, we wanted to determine the per-instruction power profile for the POWER architecture. Work done by Tiwari et. al. [1] on instruction-level power and its applications suggests that providing a compiler with per-instruction power data is an important step towards the generation of low-power executables. The paper describes a direct current measurement technique for determining power consumption. Due to resource constraints, we decided to use the POWER architecture simulator rather than real hardware for determining per-instruction power.

Pipeline gating work done by Manne et. al. [2] inspired us to consider speculation control as another avenue for power reduction. This technique tries to avoid unnecessary work caused by the speculative execution of instructions at branches that are hard to

predict. Speculation control was achieved through gating the fetch stage of the pipeline. It is estimated that flushed instructions account for over 6% of the total energy consumed in an Alpha 21264 [3]. We planned to extend speculation control work into an optimization technique that could be used at compile time, thereby avoiding any changes to the hardware. If the hard-to-predict branches could be determined at compile time, these branches could be augmented with low-power instructions such as NOP. Doing so would ensure that the processor does not execute high-power instructions that are eventually flushed once the branch is resolved. The performance loss of this technique should be negligible as speculative execution is not helpful at those branches.

Reducing switching activity is another area that has received a lot of research interest. To minimize the switching, earlier research proposed techniques such as using grey code for memory addresses [4], better instruction scheduling methods [5, 4], efficient number representations [6] and relabeling registers [7]. We planned to further study instruction re-ordering as a means to minimize the activity factor. Specifically, we also wanted to determine whether there are certain commonly occurring instruction sequences in workloads that consume high power. Finding a low power alternative such as, transformation to an equivalent yet low-power instruction sequence would help reduce power. Knowledge from this research would be used to incorporate power-oriented optimization options into an existing compiler such as *gcc*.

## 2. Experimental Infrastructure

We use the Turandot/PowerTimer simulator from IBM for studying the POWER architecture and a direct measurement technique for the Alpha architecture. The details of each is described below:

**Turandot/PowerTimer:** Turandot is a microarchitectural performance simulator for the POWER architecture. The flexibility and inherent speed of the simulator allows for a wide-scale design space exploration. Though primarily a trace-driven simulator, it can be used in the execution-driven mode using Aria which dynamically generates traces under Turandot’s control. Aria however is supported only on AIX. A power model was later added to Turandot and the simulator was called Turandot/PowerTimer. Power models derived from circuit level simulations of POWER4-like circuits are cou-

pled with the microarchitectural sub-unit utilization data from the cycle-accurate simulator to estimate the power consumption. The simulator does not capture bit-level activity factor as it is believed to account for only a small percentage of the total dynamic power consumption. We discovered this through experimentation. However, we expected the simulator to model this since we were studying power consumption at the instruction level.

**Direct Measurement:** This setup was used for studying the power consumption of a Alpha 21264 processor. A precision current-sense resistor ( $0.005\Omega$ ) was inserted in the processor power supply line of a Compaq AlphaServer DS10 motherboard. The voltage drop across the resistor measured using a digital multimeter, was used to determine the current flow and hence the power consumption of the processor. Standard off-the-shelf Linux (kernel 2.4) and HP Tru64 operating systems were studied. These OSes were started in single-user, non-graphical mode.

## 3. Initial Experiments

**Per-instruction power:** During the code generation phase, optimizing compilers often select instructions that achieve the best performance. However, this does not always results in a least-power code [1]. Providing the compiler with per-instruction power data will enable it to pick instruction that are power economical. In order to determine per-instruction power, the PowerTimer simulator was modified to keep track of the number of cycles and the difference in cumulative core power between instruction fetch and retire. Using this, the average core power for the duration of the instruction execution can be determined. Apart from the instruction of interest, the total core power is also affected by other in-flight instructions which can be in excess of 200 for the POWER4 architecture. Therefore, synthetic traces with the desired instruction embedded in a stream of NOPs was generated. This ensures a consistent pipeline state when the instruction of interest is fetched. Table 1 shows the average power consumed for a few instructions of the POWER architecture. Since the

Instruction	Cycles	Power (W)
ADD, SUB	14	25.77
FMUL	18	25.11
FDIV	41	23.38
LWZ	16	25.92
B	13	25.66

Table 1: Per-instruction power consumption for the POWER architecture

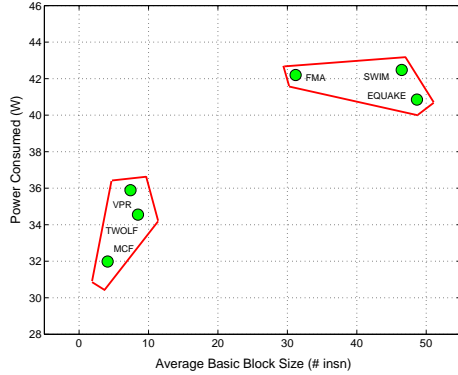


Figure 1: Correlation between average basic block size and power consumed

simulator does not capture bit-line activity factors, the results are unaffected by the operand values. One interesting observation that can be made from the table is that per-instruction power for floating point instructions such as FDIV is lower than that of other instructions. This is because such operations take more cycles to execute, thereby decreasing the amount of work done in each cycle.

For the Alpha architecture, synthetic executables with a single infinite loop containing multiple instances of the instruction of interest were created. The infinite loop ensures that a stable voltage measurement can be obtained from the digital multimeter. The loop body must be long enough to nullify the effect of the loop-terminating branch instruction and at the same time should be short enough to avoid excessive i-cache misses. Table 2 shows the per-instruction power measurements for the alpha architecture. Among the instructions studied, the NOP instruc-

Instruction	Power (W)
NOP	49.40
BGE	51.23
ADDQ	55.22
STL	59.31
LDQ	60.06

Table 2: Per-instruction power consumption for the Alpha architecture

tion was found to consume the least power while the load quadword memory instruction, LDQ, consumed the most power. Another interesting observation made through direct measurements is that techniques such as minimal distance instruction encoding have very little effect on the overall power consumption. For example, the difference in power consumption between code sequences with zero and max-

imal inter-instruction bit transitions was observed to be just 0.08W. Control over inter-instruction bit transitions was done through the register fields. Bit transitions on data had a more prominent effect on power. For example, a difference of 0.2W in power consumption was observed between two instruction sequences of add instructions that worked on the same registers but had different contents.

**Code, power characteristics:** For the SPEC traces provided by IBM, relationships between code characteristics and power profiles were examined. For each benchmark, a strong correlation between average basic block size and power consumed was observed. Figure 1 shows this correlation. The basic-blocks of the benchmarks in the upper cluster of Figure 1 are approximately 4 times larger than those in the lower cluster. Also, the branch mis-prediction rate of the upper cluster was half that of the lower cluster. This causes the benchmarks in the upper cluster to less frequently flush the pipeline thereby increasing utilization of the various sub-units of the processor resulting in a larger power consumption. We also analyzed the instruction-mix in regions of benchmark execution where large amounts of power was dissipated. These regions are typically characterized by load, store and floating-point multiply instructions.

**OS power optimizations:** The direct measurement technique on the 21264 was used to study the idle state power consumption of the Linux (kernel 2.4) and HP Tru64 operating systems. Idle state power is the power dissipated by the processor when the OS finds no processes ready for scheduling. This scenario is often encountered in systems predominantly running I/O bound jobs such as email and web-browser wherein, the processes are often blocked waiting for I/O. Laptops are typical examples of such systems. Reducing idle state power will prolong battery life of such devices without compromising performance. Under idle state, the Linux OS dissipates 56.54W which is 3.3W lower than that of Tru64 (59.84W). This difference in power can be attributed to the operations performed by the OS in idle state. The code executed by Linux under idle condition is shown below:

```

$idle:
    ldq    $1, 0($9)
    bge    $1, $idle # No ready process
    .. invoke scheduler

```

The idle loop in Linux does a memory read which indicates if any process is in ready state. This loop exe-

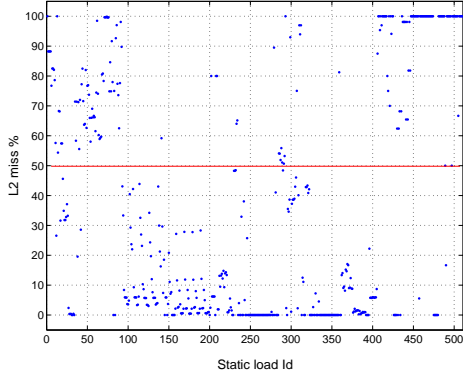


Figure 2: L2 miss rate for static memory references for the *mcf* benchmark.

cutes indefinitely until a ready process is found. From table 2, the arithmetic mean of the power consumed by the LDQ and BGE instructions is found to be 55.64W which is close to the idle state power of the Linux OS. LDQ is a high power instructions and therefore, decreasing its frequency of execution will considerably decrease idle power. This was done by inserting several NOP instructions in the idle loop, which resulted in 6.75W reduction in power consumption. The only down side to this modification is that scheduler activation is delayed by a few cycles after any process moves to ready state. This however will be hardly noticeable due to the relatively slow speed of I/O activities. For CPU-bound jobs, no performance degradation was observed.

#### 4. Future Research Direction

The first phase of our research mainly focused on gaining familiarity with the research infrastructure. The next phase will involve research pertaining to compiler optimization to conserve power. To achieve this, we plan to investigate the following:

(1). Controlling speculative execution at hard-to-predict branches. First, we will quantify the amount of power wasted due to the execution of mis-speculated

instructions for the POWER architecture. Next, a technique to identify hard-to-predict branches at compile time will be identified. Power saving is achieved by augmenting these branches with low-power instructions such as NOPs.

(2). Making the compiler cache aware. This is motivated by the fact that certain static memory references in an executable exhibit high cache miss rates. Figure 2 shows the L2 miss-rate of each static memory reference instruction that caused a L1 miss for the *mcf* benchmark. It is observed that 36% of the static references have a >50% L2 miss rate. We want to focus on techniques such as compiler directed prefetching, load re-ordering and data locality improvements to reduce the power expended due to cache misses.

#### References

- [1] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software, 1996.
- [2] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. *SIGARCH Comput. Archit. News*, 26(3):132–141, 1998.
- [3] Karthik Natarajan, Heather Hanson, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Microprocessor pipeline energy analysis. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 282–287. ACM Press, 2003.
- [4] Ching-Long Su, Chi-Yang Tsui, and Alvin Despain. Low power architecture design and compilation techniques for high-performance processors. In *Proceedings of IEEE COMPCON*, 1994.
- [5] Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler optimization on instruction scheduling for low power. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 55–60. IEEE Computer Society, 2000.
- [6] John Sacha and Mary Jane Irwin. Number representations for reducing data bus power dissipation. In *Proceedings of ICASSP*, 1998.
- [7] W. Ye, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Design Automation Conference*, pages 340–345, 2000.