

The Effect of Compiler Optimizations on Pentium 4 Power Consumption

John S. Seng

Dean M. Tullsen

Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{jseng,tullsen}@cs.ucsd.edu

Abstract

This paper examines the effect of compiler optimizations on the energy usage and power consumption of the Intel Pentium 4 processor. We measure the effects of different levels of general optimization and specific optimization. We classify general optimizations as those compiler flags which enable a set of compiler optimizations. Specific optimizations are those which can be enabled and disabled individually. The three specific optimizations we study are loop unrolling, loop vectorization, and function inlining. The binaries used in this study are generated using the Intel C++ compiler, which allows fine-grained control over each of these specific optimizations.

1. Introduction

The power consumption of general purpose microprocessors has reached a point where the problem has to be addressed at various levels of system design. Many circuit, architecture, and software algorithm techniques exist to reduce power, but one often overlooked area is the effect of the program code on power consumption. Some research has been done studying the effect of compiler optimizations on power consumption [8, 9]; this work has been generally limited to using architecture-level power models for power estimation. In this work we examine the effect of the compiler on the energy usage and power consumption of an actual modern processor, the Intel Pentium 4 processor.

Given a particular architecture, the programs that are run on it will have a significant effect on the energy usage of the processor. The manner in which a program exercises particular parts of the processor will vary the contribution of individual structures to total energy consumption. For example, if the execution of a particular program generates a significant number of data cache misses, the energy used by the second level cache will increase, as there will be more access to the secondary cache.

The relative effects of program behavior on processor energy and power consumption can be demonstrated in simulation, but not until real systems are tested can the absolute values be known. There are additional effects that are difficult to accurately simulate in architecture-level simulation. One example is the power consumption of clock generation and distribution. The energy consumed by the clocking network is a significant one and is difficult to quantify in architecture-level simulation [2]. An additional source of energy usage is that of leakage. Leakage exists even when the processor is not performing any useful computation. Leakage will be a more significant fraction of overall power consumption as supply voltages are scaled down [1]. Again, leakage is a factor that is difficult to quantify in architecture-level simulation. Given these difficulties, it is necessary to study the effect of programs on a real processor and not just in simulation.

In this study, we measure the energy and power consumption of a 2.0 GHz Intel Pentium 4 processor. We run different benchmarks compiled with various optimizations and quantify the energy and power differences when running different binaries. In our experiments we use a modern compiler, the Intel C++ compiler, to produce the binaries. This compiler contains many optimizations, including those that specifically target the capabilities and features of the Pentium 4 processor.

This paper is organized as follows. Section 2 describes prior research related to this work. Section 3 describes the power measurement platform along with the benchmarks tested. Section 4 examines the results of using the various compiler optimizations. Section 5 describes future work. Section 6 concludes.

2. Related Work

Most previous work studying compiler optimizations on power consumption has involved architecture-level power models coupled with cycle-by-cycle performance simulators. Valluri et al. [9] study the effect of compiler opti-

mizations on the power consumption and energy usage of an out-of-order superscalar processor. They simulate a processor using the SimpleScalar simulator coupled with the Wattch [5] architecture level power model.

Kandemir et al. [8] study the effect of various compiler optimizations on the energy and power usage of a processor core and memory subsystem. The work is targeted towards embedded applications. Their work is based on SimplePower [10], a cycle-by-cycle architecture level power model. The work does not consider more advanced optimizations available on general purpose processors such as loop vectorization or interprocedural optimizations.

Chakrapani et al. [6] also present a study into the effect of compiler optimization on the energy usage of an embedded processor. Their work targets an ARM embedded core and they use an RTL level model along with Synopsys Power Compiler to estimate power.

Other prior related work has been done in measuring the power consumption of processors while a system is running. Joseph et al. [7] measure the power of Pentium Pro processor while it is running various benchmarks. In that work, the entire system power consumption is being measured and not just the processor power consumption.

3. Methodology

The power measurement platform we use consists of a 2.0 GHz Intel Pentium 4 Processor running on a Gigabyte GA-8IEXP motherboard. The system is running Redhat Linux 7.3 with a uniprocessor 2.4.18 kernel. The kernel is from a stock Redhat installation. The system has 1GB of PC2100 DDR memory. During power measurement the operating system is in console mode (i.e. X-windows is not running) and there are no other actively running processes on the machine. All results reported are for the energy and power of the microprocessor alone and for no other components.

In order to measure the current consumed by the running system, we insert two 5 watt .015 Ohm resistors in-line with the V_{CC} trace to the processor core. The voltage drop across the resistors is measured at run time and the current to the microprocessor is computed given the known resistance value. The resistors are connected in parallel allowing for a total power dissipation of 10 W. During our experiments, the resistors dissipated up to 7 W. Although there is a voltage drop across the resistors, the voltage to the processor is regulated by the motherboard to stay at the required operating voltage. The V_{CC} trace is from the voltage regulator module (VRM) of the motherboard and runs directly to the processor. This trace is used solely by the processor (it is not used to power any other components in the computer). The processor core runs at 1.5 volts. During our experiments, the current through this trace ranged from 7 to 30 amps. When the processor was completely

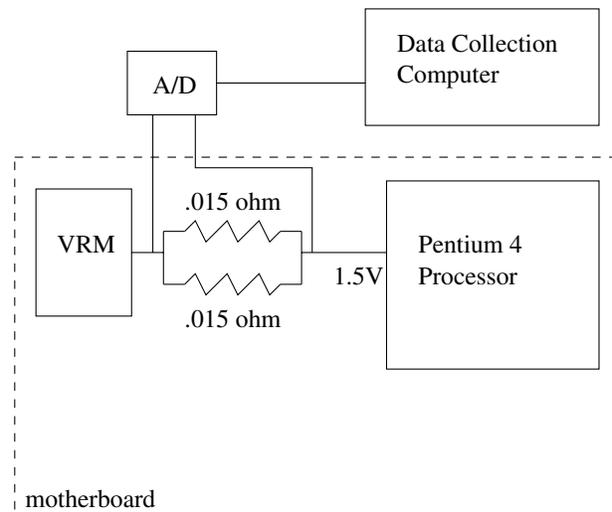


Figure 1. Power measurement platform

idle, the power dissipated was 10.5 watts. We compute the energy utilized by multiplying the current times the sample interval time and sum these products over all samples for a given benchmark run.

Analog to digital conversion is achieved using a Texas Instruments ADS1210 converter clocked at 10 MHz. We use this converter to sample the voltage across the resistors at 100 Hz. When sampling at this rate, the converter provides 22 bits of resolution. Voltage readings allows us to compute the current draw of the processor and hence, the power consumption. A single board computer is used to transmit the digital values to a data collection computer via the serial port. A schematic of the measurement setup is shown in Figure 1.

The benchmarks we use for our experiments are from the SPEC 2000 benchmark suite. The integer benchmarks are *crafty*, *bzip2*, *gcc*, *mcf*, and *parser*. The floating point benchmarks are *ammp*, *art*, and *equake*. Power measurements are taken from the beginning of the benchmark run to completion. Each result shown represents an average of 3 consecutive runs of the benchmark. For a given benchmark, the variation of total energy during the 3 runs ranged from .5% (best case) to 1.0% (worst case).

We use one compiler in this study. The compiler is the Intel C++ compiler v.6.0.1 for Linux. This is an advanced compiler featuring many levels of optimization. Each of the flags used in the study are documented in [4]. The compiler provides support for the additional x86 instructions available on the Pentium 4 and contains optimizations targeting the Pentium 4 pipeline.

4. Results

We categorize the flags controlling the compiler optimizations as either general or specific. The general opti-

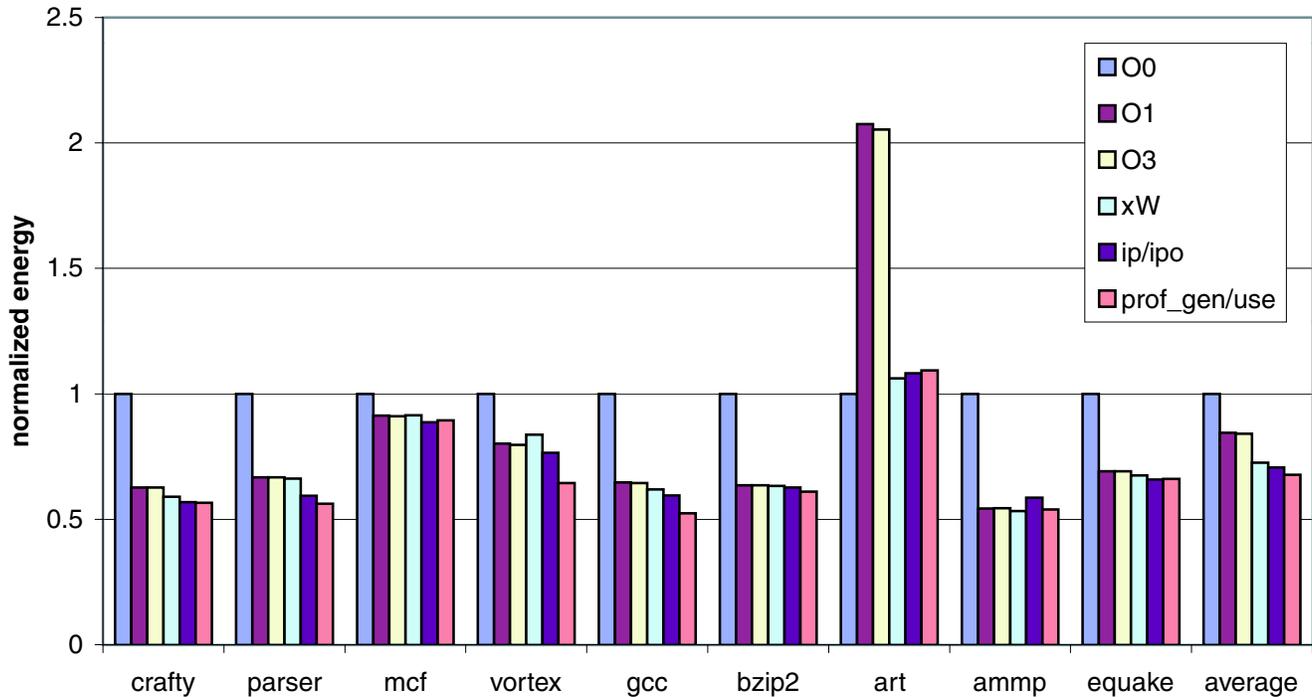


Figure 2. Normalized total energy usage

mization flags are those flags which enable a set of compiler optimizations. These flags are used to simplify compilation by enabling a group of optimizations with just one flag. The specific optimization flags are those which enable or disable one particular optimization. These flags are included in order to allow the user to selectively turn on or off a particular optimization, even when other general flags are included. We study the effect of these two types of flags separately.

4.1. General Optimizations

For the study of the effect of general optimizations, the following flags are used:

-O0 This flag disables all compiler optimizations.

-O1 This flag enables the following optimizations: global register allocation, instruction scheduling, register variable detection, common subexpression elimination, dead-code elimination, variable renaming, copy propagation, constant propagation, strength reduction-induction variable, tail recursion elimination, and software pipelining. **-O2** enables the same optimizations as **-O1**.

-O3 Enables the optimizations in **-O1** and adds: prefetching, scalar replacement, and various loop transformations.

-xW Enables code generation of MMX technology instructions, Streaming SIMD Extensions, and Streaming SIMD Extensions 2. These instructions are SIMD operations that are available on the Pentium 4 Processor [3].

-ip/-ipo Enables interprocedural optimizations within a file and across multiple files. These optimizations include: function inlining, constant propagation, and dead code elimination.

-prof_gen/-prof_use The **-prof_gen** flag generates binaries which produce profiling information regarding common branch targets, function call frequency, and dynamic number of loop iterations. When the sources for the benchmarks are recompiled using the **-prof_use** flag, the compiler produces optimized binaries given the profiling information.

In this section, when a result is given for any one of the general optimizations, all previous flags are also enabled (except if **-O3** is enabled, **-O0** and **-O1** are not enabled). For example, if a result is given for the **-xW** flag, the **-O3** flag is also enabled.

Figure 2 shows the total normalized total energy used by each of the benchmarks and for the average of the benchmarks. The results for each of the benchmarks is normalized to the energy used when the benchmark is run with all optimizations disabled.

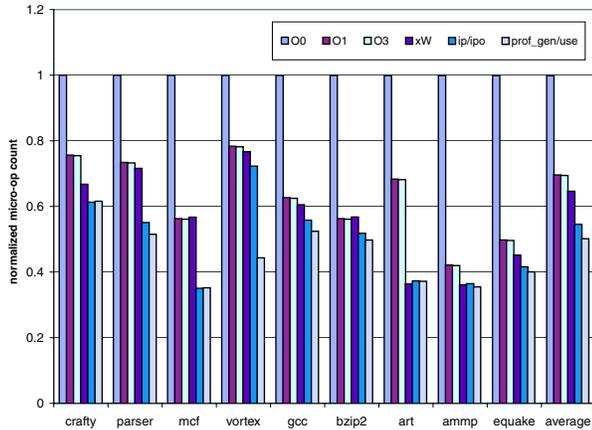


Figure 3. Retired micro-op count

For almost all benchmarks, significantly less energy is required when optimizations are enabled. For all of the benchmarks, there is very little energy difference between the `-O1` and `-O3` optimization flags. One benchmark, `art`, exhibits a behavior different from all the others when the optimizations are applied. When the first two levels of optimization are applied, the run time of `art` is almost doubled. We found that this occurs because of an increased number of branch mispredictions when the `-O1` and `-O3` flags are enabled. The number of branch mispredictions increased by more than 85% in both cases. The average energy reduction across all benchmarks when enabling the `-O1` flag is 15.5%. When `art` is not included in the average, the energy reduction is equal to 30.9%.

The `-xW` and `-ip/-ipo` optimizations each provide additional energy savings. Again, the `art` benchmark behaves in an anomalous manner, with increasing energy as the optimizations are applied. When `art` is not considered in the average, the `-xW` and `-ip/-ipo` flags reduce energy by an additional 0.7% and 2.2%, respectively.

The average energy reduction with profiling information and full optimizations is 32.3%, compared to binaries with no optimization. The profiling optimization is quite beneficial to `gcc` and `vortex`. Compared to `-ip/-ipo`, those benchmarks produced with the profiling information incur fewer branch mispredictions (3.4% for `gcc` and 9.3% for `vortex`). Additionally, figure 3 shows the normalized retired micro-op count for each of the benchmarks. The retired micro-op count for the profiled binaries is 3.3% and 27.1% less than `-ip/-ipo` for `gcc` and `vortex`, respectively.

A comparison of the total energy used and the micro-op count show that the two are highly correlated. The two cases where there is not a strong correlation is with `art` (previously discussed) and `mcf`. Although the retired micro-op count for `-ip/-ipo` and `prof_gen/use` for `mcf` is reduced in each case by over 20% compared to `-xW`, the run

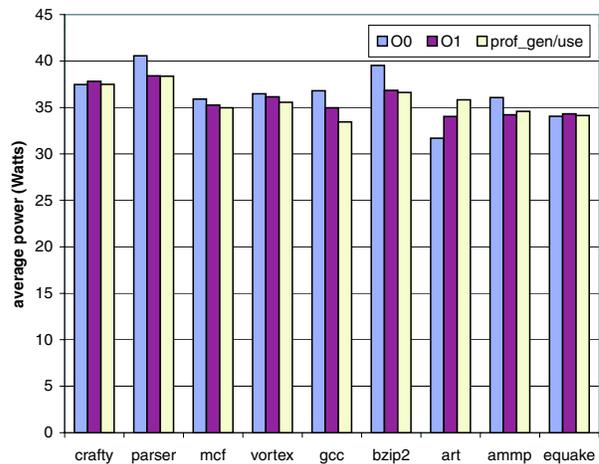


Figure 4. Average processor power

time is reduced only by 1%. The run time of `mcf` is dominated by cache misses and we find that the latter two optimization flags reduce cache misses by only 0.1% (over `-xW`).

Although this data demonstrates that each of these optimizations reduces the total energy used by the processor, we find that the energy used by the program is even more closely correlated with run time. When comparing the run time of the benchmarks with the total energy used, we find that there is a nearly linearly correlation with energy consumption. In addition, over 10 watts of power is dissipated when the processor is idle. This power is dissipated in part because of the factors discussed earlier, clocking and leakage. Given that an idle processor dissipates this power, the energy differences seen because of code sequences is greatly reduced. Because of these factors, for the Pentium 4 processor, we conclude that compiling for the best performance leads to the most energy-efficient binaries.

Figure 4 shows the average power consumption of the processor when running each of the benchmarks. The results shown are for all optimizations disabled, the default optimizations enabled, and all optimizations enabled (including interprocedural and profiling optimizations). The average power consumption of the benchmarks ranges from 32 watts to 41 watts. Although the optimized binaries perform a similar amount of work to the unoptimized binaries and in a shorter period of time, the average power consumption of the optimized binaries is typically lower. Executing the optimized binaries often results in executing fewer instructions overall. Optimizations such as loop vectorization while using functional units which consume more power, actually reduce the number of instructions required to do the same computation. This leads to reduced power of the units throughout the machine, such as the fetch unit and issue mechanism.

4.2. Specific Optimizations

We categorize specific compiler optimizations as those optimizations which can be enabled and disabled individually via command line flags (without affecting the status of other optimizations). An analysis of these optimizations allows us to study the effect of particular code generation techniques on energy and power consumption. The specific compiler optimizations studied are the following: loop unrolling, loop vectorization, and function inlining. We list the name of the optimization along with the flags that the compiler accepts in order to disable the particular optimization.

loop unrolling (*-unroll0*) By default the loop unrolling optimization is enabled during compilation. Heuristics are used by the compiler to determine when to perform loop unrolling and the number of times to unroll a loop. Although the various parameters for loop unrolling can be specified at compile time, we use the default heuristics when unrolling is enabled. The benefit of loop unrolling is that the generated code generally requires fewer dynamic instructions; the trade-off is that the code size is generally larger (for the average of the benchmarks examined, the size increase is 1.5%) and possibly additional instruction cache misses.

loop vectorization (*-vec-*) Loop vectorization is an optimization where the code generated for loops use the MMX, SSE, and SSE2 instructions. The compiler detects when loops can be parallelized via these SIMD instructions.

function inlining (*-ip_no_inlining/-ip_no_pinlining/-no_lib_inline*) These flags when used with the *-ip* flag disable inlining of functions. The function inlining optimizations insert the actual code sequences of function calls instead of performing the function call.

In order to quantify the results of a particular optimization, we compare the energy, power, and execution time of a binary with the optimization disabled with that of a binary produced using the *-O3*. This optimization flag produces optimized binaries using all of the optimizations we study. The reference binary for the function inlining experiment is compiled with the *-ip* and *-ipo* flags enabled.

Figure 5 shows the results for the average of the benchmarks when each of the optimizations is enabled. The data represents the percentage reduction for the metric when the optimization is applied. The data for each of the individual benchmarks is shown in Table 1. Negative values indicate an increase in the metric.

For the average of the benchmarks, loop unrolling proved to have little effect in terms of program execution time. Some benchmark performance was particularly aided

crafty	energy	time	power
unrolling	1.68	0.39	1.29
vectorization	6.94	6.99	-0.05
inlining	6.33	4.77	1.49
parser	energy	time	power
unrolling	0.45	0.46	-0.01
vectorization	0.12	0.08	0.04
inlining	8.50	8.85	-0.32
mcf	energy	time	power
unrolling	-0.23	-0.18	-0.05
vectorization	0.58	0.63	-0.05
inlining	1.36	1.07	0.29
vortex	energy	time	power
unrolling	2.66	2.45	0.21
vectorization	-3.72	-3.25	-0.49
inlining	3.98	4.57	-0.56
gcc	energy	time	power
unrolling	-2.45	-2.97	0.53
vectorization	-1.17	-1.51	0.34
inlining	4.01	4.39	-0.36
bzip2	energy	time	power
unrolling	0.61	-0.01	0.62
vectorization	-2.34	-1.82	-0.53
inlining	1.33	1.19	0.14
art	energy	time	power
unrolling	-2.24	-1.95	-0.29
vectorization	0.66	0.68	-0.02
inlining	0.29	0.42	-0.14
ampp	energy	time	power
unrolling	1.79	1.61	0.18
vectorization	-0.13	-0.03	-0.10
inlining	16.80	14.89	1.66
quake	energy	time	power
unrolling	0.67	0.15	0.52
vectorization	-0.01	-0.94	0.94
inlining	1.96	1.28	0.66

Table 1. Percent reduction when optimization is enabled.

by unrolling while for others performance was reduced. The optimization was most beneficial for *vortex* and *ampp*. When applying loop unrolling, there is a slight measurable reduction in energy, for little or no effect on performance. For the binaries where loop unrolling enabled, the total energy is reduced as well as the power consumption. The difference in terms of energy and power is very small, though. When compiling for low energy, whether or not loop unrolling should be enabled depends specifically on the program.

Loop vectorization and function inlining behave similarly. On the average, both optimizations reduce total energy consumption, but with a corresponding reduction in execution time. Each optimization provides benefits on certain benchmarks, but degrades the performance of others. Loop vectorization does well on *crafty* (6.94% reduction in energy) and function inlining does particularly well on *ampp* (16.80% reduction in energy). As power and performance optimizations, neither optimization demonstrates

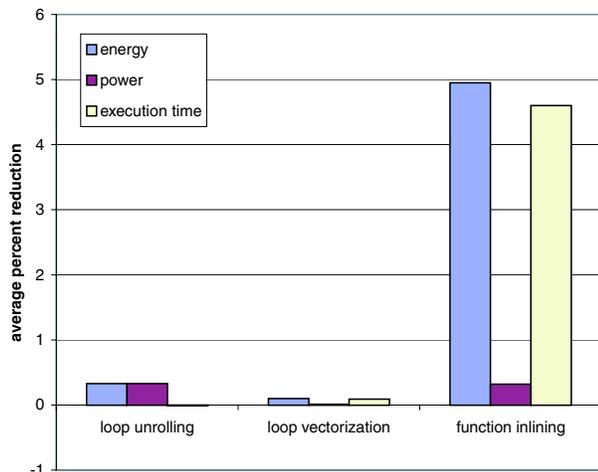


Figure 5. Average percentage reduction for individual compiler optimizations.

any sort of positive trade-off between providing high performance and low energy usage.

Of these optimizations, one cannot conclude that there is any which can be classified as an optimization specific to compiling for energy/power minimization. We find that similar to the general optimizations, compiling for the lowest energy usage involved selecting the flags which provide the shortest execution time. Additionally, compiling for low power consumption also generally means compiling for shortest execution time. A few benchmarks, such as *parser* and *vortex* with inlining, consume more power with optimizations enabled; but the effects on power are negligible. In general though, the effect on power consumption follows that of energy and execution time.

5. Future Work

Additional work that we would like to perform would be to analyze what type of instructions or instruction sequences lead to increased power consumption. Future work would involve quantifying the effect of particular instruction sequences on energy and power consumption. We would also like to examine the effect of compiler optimizations on various other processors, such as RISC or VLIW processors.

We found that for the processor we studied, there were no individual optimizations which provided significant energy/power savings without reducing program run time. We believe that given an architecture which exposes more architecture control to the software, opportunities will exist for more power/performance optimizations.

6. Conclusion

In this work, we measure the effect of various compiler optimizations on the Intel Pentium 4 processor. We use a modified motherboard to measure the current draw of a microprocessor running binaries compiled with various optimizations. The compiler used in this study is the Intel C++ compiler v.6.0.1.

We find that enabling various general compiler optimizations leads to significantly reduced energy consumption of the processor. Although the reduction in energy is significant, the reduction comes primarily from a reduction in program execution time.

Additionally, we examine the effect of individual optimizations on energy usage and power consumption. The optimizations studied are loop unrolling, loop vectorization, and function inlining. For the benchmarks studied, loop unrolling provides a very minor benefit in terms of reducing energy; loop vectorization and function inlining demonstrate no advantage for performance to power optimization.

We find that for the Pentium 4 processor, the most effective approach to compiling for low energy and power is to compile for the highest performance. We find that this is true when the goal is either to minimize energy or minimize power consumption.

References

- [1] Mos scaling: Transistor challenges for the 21st century. In *Intel Technology Journal*, 1998.
- [2] Cool chips tutorial: an industrial perspective on low power processor design. 1999.
- [3] *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, 2002.
- [4] *Intel C++ Compiler User's Guide*. Intel Corporation, 2002.
- [5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.
- [6] L. Chakrapani, P. Korkmaz, V. Mooney, K. Palem, K. Puttaswamy, and W. Wong. The emerging power crisis in embedded processors: What can a (poor) compiler do? In *CASES 2001*, Nov. 2001.
- [7] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. In *The International Symposium on Low Power Electronics and Design*, Aug. 2001.
- [8] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, July 2000.
- [9] M. Valluri and L. John. Is compiling for performance == compiling for power? In *Proceedings of the 5th Annual Workshop on Interaction between Compilers and Computer Architectures*, Jan. 2001.
- [10] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *27th Annual International Symposium on Computer Architecture*, June 2000.