A Fine-Grained Analysis of the Performance and Power Benefits of Compiler Optimizations for Embedded Devices [*]

Jason W.A. Selby and Mark Giesbrecht
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada, N2L 3G1
{j2selby,mwg}@uwaterloo.ca

ABSTRACT

In this study we examine the performance and power benefits resulting from many traditional and aggressive compiler optimizations. Examples of early and loop optimizations were translated from classic compiler textbooks into PowerPC assembly and executed in the Dynamic SuperScalar Wattch simulation environment. Each optimization was applied in an isolated manner which resulted in an overall average improvement in performance of 4.8% and a 6.2% decrease in power consumption for the early optimizations. The loop optimizations yielded greater performance and power improvements with an average speed-up of 17.0% and a 15.3% reduction in power consumption. The decrease in power consumption of the early optimizations was found to correspond closely to the execution time. In contrast, the improvements from the loop optimizations were found to correspond to the number of instructions committed. The average decrease in the number of instructions committed was found to be 6.2% and 14.1% for the early and loop optimizations, respectively. The goal of this study is to provide guidance to compiler writers when developing methods to select optimizations if power consumption is of importance.

1. INTRODUCTION

This study performs a comparison of the performance and power benefits resulting from many traditional and aggressive compiler optimizations. We translate representative examples from an extensive list of what are now standard static compiler optimizations into PowerPC assembly code [7] (as derived from [8]). The unoptimized and optimized assembly code sequences are then executed in the Dynamic SuperScalar Wattch (DSSWattch) simulation environment to capture their performance and power characteristics.

Previous studies [10, 4, 9] have examined the impact of compiler optimizations on power consumption in resource-constrained embedded devices. However, in general their examinations focused on the sets of transformations applied at various levels of optimization (for example, the optimizations applied by gcc at `-O3`). This paper presents a fine-grained study of compiler optimizations in isolation that is achieved by hand programming various transformations in PowerPC assembly. Specifically, we examine the results obtained by the *early optimizations* of constant propagation, constant folding, copy propagation, dead-code elimination, if-simplification, inlining, and value numbering. Additionally, the more aggressive *loop optimizations* of bounds-checking elimination, loop-invariant code motion, unrolling, unswitching, fusion, interchange, skewing, and tiling are also studied. All optimizations are applied in isolation, except constant folding, which is naturally performed after constant propagation.

The focus of optimizing compilers targeting embedded processors has traditionally been on improving performance and minimizing the size of the binary. In the past, typically, most applications were written in a mix of C and assembly code upon which exhaustive optimization was applied by a superoptimizer given unlimited time and computing resources. The increased processing power and memory capacity of current generation embedded devices has coincided with the coming of age of dynamic compilers, which has fostered a shift toward Java as the language of choice for application development targeting embedded devices.

The contribution of this study is that it serves as a metric for deciding which optimizations should be performed at different optimization levels if power is of equal concern as performance. The selection of the appropriate optimizations for the various levels is important in an adaptive dynamic compilation environment such as [3] which applies increasingly aggressive optimizations at each successive level. It is feasible that static optimizing compilers which target the embedded market will ship with power specific optimizations which are enabled via a command line switch analogous to performance optimizations (i.e., `-P3`).

The remainder of the paper is organized in the following manner. In Section 2, we discuss related research that has compared the impact of compiler optimizations on both performance and power. Section 3 details the compiler optimizations that are examined in this study. In Section 4, we provide an outline of the simulation environment in which the benchmarks are executed and the results of the simulations are reported and discussed. Finally, we conclude and point out promising directions from this work in Section 5.

[*]This research was supported in part by a Natural Sciences and Engineering Research Council of Canada Strategic grant.

2. RELATED WORK

Previous studies have found that compiling for performance (i.e., fastest execution time) is equivalent to compiling for the minimization of power consumption. In [10], Valluri *et al.* used the Wattch simulation environment to compare the performance and power benefits provided by the sets of optimizations applied at the various levels of the DEC Alpha C compiler (`-O0` through `-O4`). Additionally, [10] examined the influences of the specific optimizations of instruction scheduling (both basic list and aggressive global scheduling), inlining, and loop unrolling, all of which can be enabled by a command line option to gcc.

In [9] a similar study was performed, however, the focus was specifically on the power consumption of the Intel Pentium 4 processor. They examined the performance and power improvements resulting from `-O0` to `-O3` of Intel's C++ compiler. Loop unrolling, loop vectorization and inlining were also examined, as they too can be controlled via command line switches.

A categorization of compiler optimizations in respect to their effect on power consumption was proposed in [4]. They defined *Class A* optimizations as those which yield an improvement in power consumption that is directly attributable to the decreased execution time. Optimizations categorized as *Class B* either slow down or have no effect on execution time, yet they decrease power consumption. *Class C* optimizations increase the amount of power consumed irrespective of the impact on performance (however, in general, the increased power consumption is in conjunction with an increase in execution time)[4].

3. BENCHMARKS

This section briefly describes each of the early and loop optimizations examined and the resulting performance and power benefits. The examples from [8] were selected since they represent the prototypical application of an optimization and therefore achieve a close approximation of the derivable benefit. Tables 1 and 2 highlight the key benefits to execution time and power consumption of the early and loop optimizations, respectively.

4. EXPERIMENTAL RESULTS

The primary goal of this study is to perform a fine-grained comparison of the effects of individual optimizations on resource-constrained devices. To this end, we first carefully describe the simulation environment in which our experiments were performed. The performance and power measurements captured by the simulation environment for each of the optimizations described in the previous section are then reported. Finally, we conclude with an analysis of the findings.

4.1 Methodology

The performance and power profiles reported in this study were captured using the DSSWattch [5] simulation environment. DSSWattch is the most recent layer on top of a sophisticated and established set of research tools for computer architecture simulation. At the base of this environment is SimpleScalar [2], a cycle accurate out-of-order simulator for the Alpha and PISA architectures. Wattch [1] introduced the ability to track the power consumption of programs executed in the SimpleScalar environment. Dynamic SimpleScalar (DSS) [6] is a PowerPC port of SimpleScalar (specifically, the PowerPC 750) that also added the capability to simulate programs which are dynamically compiled. Finally, DSSWattch is an extension of the original Wattch power module which adds floating-point support and allows for the mixed 32 and 64-bit modes present in the PowerPC architecture.

Wattch incorporates four different power models:

1. *Unconditional Clocking:* Full power is consumed by every unit each cycle.
2. *Simple Conditional Clocking (CC1):* A unit which is idle for a given cycle consumes no power.
3. *Ideal Conditional Clocking (CC2):* Accounts for the number of ports that are accessed on a unit and power is scaled linearly with the number of ports.
4. *Non-Ideal Conditional Clocking (CC3):* Models power leakage by assuming that an idle unit consumes only 10% of its maximum power for a cycle in which it is inactive.

All of the tests were performed on a 1.9GHz dual-processor AMD Opteron, running Gentoo GNU/Linux with 2GB of main memory. However, the simulation environment is that of a PowerPC 750 running GNU/Linux using the default DSSWattch configuration. Under this configuration, DSSWattch was found to be capable of simulating an average of approximately 470K instructions per second in single user mode.

4.2 Results

Our findings for the loop optimizations indicate that the improvements in power consumption are directly linked to the speed-ups attained. We found an average performance improvement of 17.0%, with a range from 5.3% to 44.5% and an average decrease in power consumption of 15.3%, ranging from 7.6% to 31.0% under the non-ideal power model (CC3). However, the power benefits resulting from the early optimizations displayed a closer linkage to the reduction in the number of instructions committed than the speed-up. The average decrease in execution time resulting from the early optimization was 4.8%, ranging from 15.8% to a slow down of 0.9% and the average decrease in power consumption was 6.2% with a range from 16.1% to an increase of 0.5% (CC3).

Note that the performance numbers reported in [10, 4, 9] are as a result of applying an entire set of optimizations, such as all of those that are examined in this study. Our results, in contrast, are obtained from a single application of each optimization in isolation. Each benchmark from [8] was placed inside a test harness where the code was executed 1000 times.

Figures 1 and 2 display a comparison of the normalized execution time, number of instructions committed, and the amount of power consumed between the original and optimized code sequences for the early and loop optimizations,

Optimization	Performance/Power Benefits
Constant Folding	Fewer instructions executed
Constant Propagation	Decreased register pressure, use of immediate instructions
Copy Propagation	Decreased register pressure and memory traffic
Dead-Code Elimination	Fewer instructions executed, decreased code size
If-Simplification	Fewer instructions executed, less work for branch predictor
Inlining	Elimination of call overhead, increased register pressure
Value Numbering	Fewer instructions executed, reduced register pressure

Table 1: Overview of the performance and power benefits arising from the early optimizations examined.

Optimization	Performance/Power Benefits
Bounds-Check Elimination	Fewer comparisons, less work for branch predictor
Fusion	Improved D-Cache and register usage
Interchange	Improved D-Cache and register usage
Loop-Invariant Code Motion	Fewer instructions executed, decreased register pressure
Skewing	Improved D-Cache and register usage
Tiling	Improved D-Cache and register usage
Unrolling	Fewer comparisons, branch predictor, increased register pressure
Unswitching	Fewer comparisons, branch predictor

Table 2: Overview of the performance and power benefits arising from the loop optimizations examined.

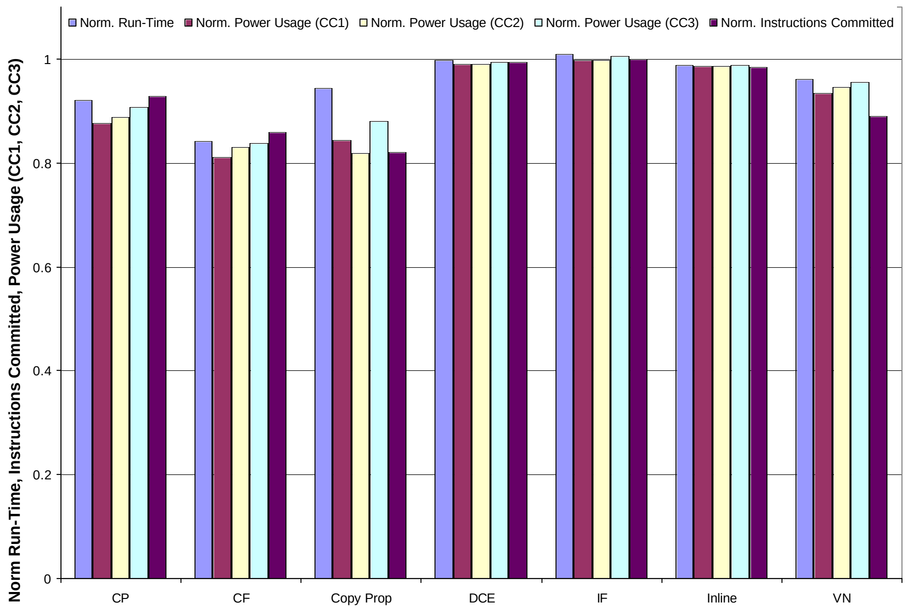

Figure 1: A comparison of the normalized execution time (optimized version divided by baseline version), the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the early optimizations.

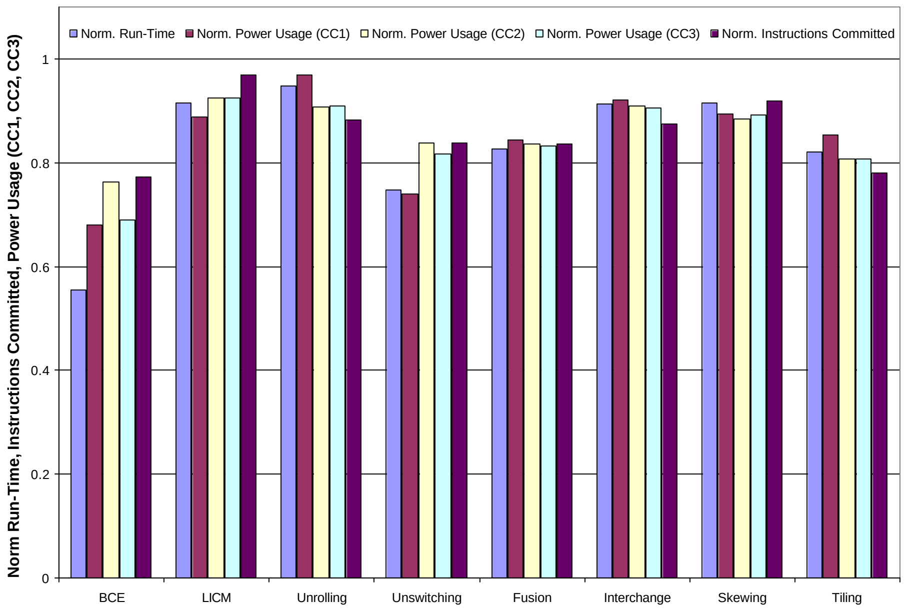

Figure 2: A comparison of the normalized execution time, the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the loop optimizations.

respectively. Power consumption for the CC1, CC2, and CC3 models is reported. However, discussion of the results is focused on the non-ideal model (CC3), since it is the most realistic incorporating the power leakage of idle units.

Out of all the early optimizations considered, the application of constant folding after constant propagation was found to have the greatest impact on performance and power providing improvements of 15.8% and 16.1%, respectively. Copy propagation and value numbering also resulted in substantial improvements, yielding a 5% minimum decrease in power consumption. Given the benefits in both performance and power, the set of early optimizations definitely performed by an adaptive dynamic optimizer should include constant propagation, constant folding, copy propagation, and value numbering. Optimizations such as dead-code elimination, if-simplification, and inlining did not contribute significantly on their own, and their applicability to a specific code segment should be guided by heuristics. It should be noted that the results obtained for function inlining do not represent the full impact that it can have. Many of the optimizations enabled after inlining has been performed were not taken into account and therefore we believe the results understate the importance of inlining.

The elimination of bounds-checking was found to provide the greatest improvement in both performance and power consumption of the loop optimizations. DSSWattch measured a speed-up of 44.5% and power savings of 31.0%. As expected, all of the loop optimizations significantly improved execution time and power usage, with a minimum return of 8.0% for each. Loop nests must always be the first place that a compiler attempts to optimize and ideally a dynamic compiler should be capable of applying all of the loop transformations examined in this study.

The variance from execution time to power consumption for the early optimizations was found to have a high correspondence. Specifically, the average variance between performance and power consumption was 0.000016%, while the variance between the number of instructions committed and power consumption was 0.002% (with standard deviations of 0.004 and 0.05, respectively). Conversely, changes in the number of instructions committed for the loop optimizations correspond more closely to power consumption than execution time, with variances of 0.002% and 0.005% and standard deviations of 0.03 and 0.05, respectively.

Tables 3 and 4 display the amount of energy consumed by each of component that DSSWattch is able to track (only the results for the CC3 power module are reported). The power savings in the load/store queue and the L2 cache were each found to be the most significant in two of the early optimizations examined, whereas, for the loop optimizations, the branch prediction unit dominated in four of the eight optimizations and the register file was the largest contributor

	CP	CF	Copy Prop	DCE	IF	Inlining	VN
Rename Unit	-6.87	-14.84	-13.99	-0.29	0.14	-1.37	-8.34
Br Pred	-4.69	-18.05	-3.30	-0.12	0.52	-1.15	-2.42
Ins Window	-9.93	-13.55	-17.00	-0.49	0.04	-1.52	-9.31
LD/ST Queue	-14.11	-17.86	-17.39	-0.58	0.48	-1.98	-10.73
Reg File	-8.38	-13.64	-8.37	-0.30	0.85	-1.00	-5.94
Int Reg File	-9.33	-9.73	469.40	-0.40	0.69	-0.67	-9.39
FP Reg File	-7.86	-15.77	-5.54	-0.24	0.94	-1.21	-3.90
L1 I-Cache	-16.83	-15.61	-20.53	-0.72	0.25	-1.60	-7.20
L1 D-Cache	-7.43	-6.24	-28.84	-1.17	0.41	-0.76	4.65
L2 Cache	-7.21	-14.38	-5.07	-0.22	0.89	-1.09	-3.57
ALU	-7.65	-15.28	-8.63	-0.34	0.74	-1.29	-5.65
Result Bus	-11.76	-14.72	-13.94	-0.66	0.29	-1.83	-4.80

Table 3: The percentage change in power consumed by each component under DSSWattch's CC3 power model resulting from the early optimizations examined.

	BCE	LICM	Unrolling	Unswitching	Fusion	Interchange	Skewing	Tiling
Rename Unit	-28.36	-9.26	-10.85	-19.15	-16.61	-14.54	-5.40	-16.61
Br Pred	-44.16	-19.42	-16.54	-28.00	-27.98	-6.16	-8.51	-3.69
Ins Window	-16.73	-3.25	-10.84	-14.93	-14.91	-9.95	-8.37	-19.52
LD/ST Queue	-21.44	-0.52	-16.48	-5.83	-23.38	-6.25	-6.34	-4.76
Reg File	-38.30	-6.98	-6.99	-20.69	-14.41	-11.60	-6.71	-20.24
Int Reg File	-26.17	-3.99	-9.30	-10.57	-9.22	-16.79	-4.10	-21.46
FP Reg File	-44.48	-8.54	-5.31	-25.29	-17.48	-8.30	-8.51	-19.53
L1 I-Cache	-19.11	-8.03	-8.95	-9.84	-16.07	2.15	-23.03	-21.08
L1 D-Cache	-41.28	-3.95	-5.54	-18.79	-13.92	-16.29	-9.62	-17.56
L2 Cache	-44.46	-8.12	-5.30	-25.22	-17.30	-8.66	-8.49	-17.75
ALU	-36.86	-6.93	-8.44	-21.52	-15.99	-9.80	-8.17	-20.90
Result Bus	-18.23	1.97	-8.38	-12.35	-15.08	-10.97	-7.60	-21.38

Table 4: The percentage change in power consumed by each component under DSSWattch's CC3 power model resulting from the loop optimizations examined.

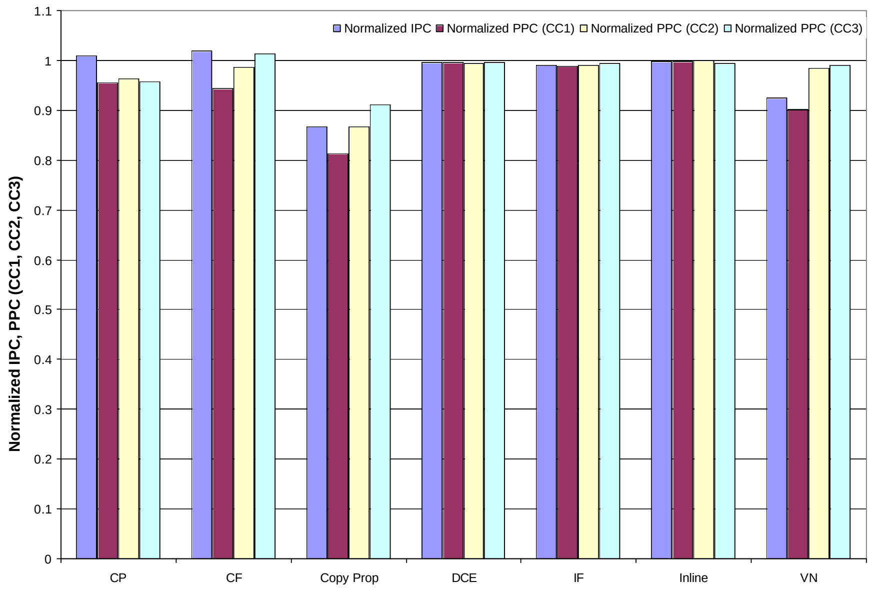

Figure 3: A comparison of the change in instructions per cycle, to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for early optimizations.

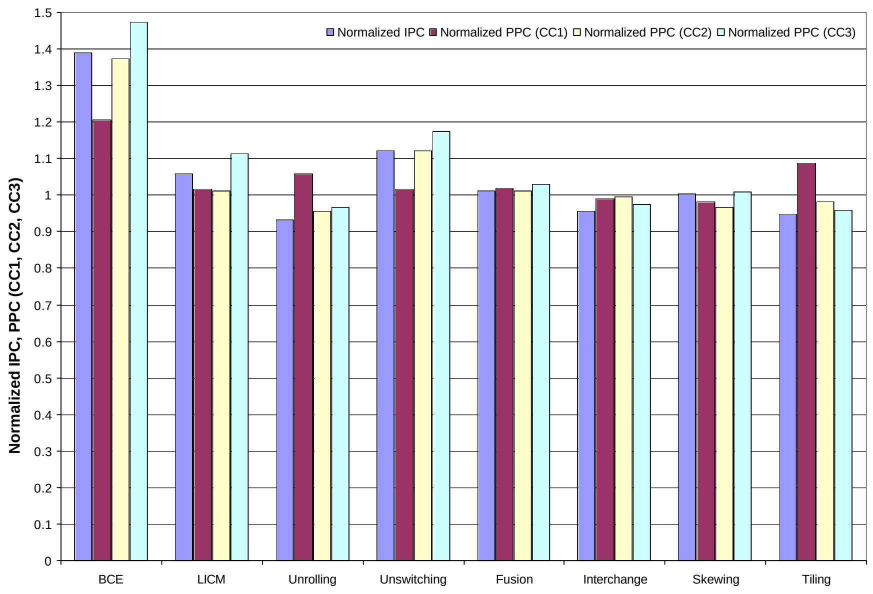

Figure 4: A comparison of the change in instructions per cycle, to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for loop optimizations.

in two of the remaining optimizations.

Although this study did not directly focus on the effects of instruction scheduling, it is interesting to examine the change in the number of instructions committed per cycle (IPC), in comparison to the change in power consumed per cycle (PPC), as displayed in Figures 3 and 4. The amount of instruction-level parallelism increased in almost all of the loop benchmarks and accordingly, so did the PPC, given that more functional units were utilized per cycle. The number of IPC were unaffected in most of the early optimizations except for copy propagation and value numbering, where the IPC was reduced by 0.14 and 0.08, respectively. In future work we intend on examining the effects of instruction scheduling in conjunction with each optimization.

5. CONCLUSION

This study examined the effects of a substantial collection of early and loop compiler optimizations on both the execution time and power consumption in the DSSWattch simulation environment. Examples of each of the optimizations were translated from [8] into PowerPC assembly and simulated by DSSWattch. Application of each optimization in isolation resulted in an overall average improvement in performance of 4.8% and a 6.2% decrease in power consumption for the early optimizations, and an average speed-up of 17.0% and average power savings of 15.3% for the loop optimizations. The improvements resulting from the loop optimizations were found to be closely tied to the decrease in the number of instructions committed, which on average was 14.1%. As a result of these findings, the earliest optimization level of an adaptive dynamic optimizer should at least include constant propagation, constant folding, copy propagation, and value numbering. Additionally, the compiler should be sufficiently sophisticated such that it can carry out the analysis required to perform all of the loop transformations examined in this study.

Unfortunately, although DSSWattch does differentiate between the amount of power consumed by integer and floating-point operations, it does not assign a cost to individual instructions. This precluded the examination of optimizations such as strength reduction, which often replaces a complex instruction sequence with a longer sequence consisting of simpler instructions. The application of strength reduction may have resulted in a performance increase, however, under DSSWattch's current power model it would have incorrectly reported an increase in power consumption, as a result of the increased instruction count. In future work the amount of power consumed by various instructions will be scaled in order to capture more accurate results.

If the findings of this study are to be applied as a criteria for the selection of the optimizations which are appropriate at each optimization level in an adaptive dynamic optimizing compiler, then the cost of performing each optimization must be studied further. In conjunction with this paper, the performance, and power impacts of applying each transformation at run-time must be factored into the level formation process. For example, if this study found that a specific optimization provided a 3% performance improvement and a 2% savings in power, however, performing the optimizations is expensive in terms of both time and power, then it may be more suitable for inclusion at a higher optimization level such as `-O3` rather than `-O1`.

6. REFERENCES

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[3] M. G. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[4] L. N. Chakrapani, P. Korkmaz, I. V. J. Mooney, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: What can a poor compiler do? In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 176–180, New York, NY, USA, 2001. ACM Press.

[5] J. Dinan and J. Moss. DSSWattch: Power estimations in Dynamic SimpleScalar. Technical report, University of Massachusetts at Amherst, July 2004.

[6] X. Huang, K. M. J.E.B. Moss, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[7] IBM Microelectronics and Motorola, Phoenix, AZ, USA. *PowerPC Microprocessor Family: The Programming Environments*, 1996. MPRPPCFPE-01.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[9] J. Seng and D. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *INTERACT-7: The 7th Annual Workshop on Interaction between Compilers and Computer Architectures*, Anaheim, CA, USA, Feb. 2003.

[10] M. Valluri and L. John. Is compiling for performance == compiling for power? In *INTERACT-5: The 5th Annual Workshop on Interaction between Compilers and Computer Architectures*, Monterrey, Mexico, Jan. 2001.