Power and Energy Impact by Loop Transformations

Hongbo Yang†, Guang R. Gao†, Andres Marquez†, George Cai‡, Ziang Hu†

† Dept of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716
{hyang,ggao,marquez,hu}@capsl.udel.edu

‡ Intel Corp
1501 S. Mopac Expressway, Suite 400
Austin, TX 78746
george.cai@intel.com

Power dissipation issues are becoming one of the major design issues in high performance processor architectures.

In this paper, we study the contribution of compiler optimizations to energy reduction. In particular, we are interested in the impact of loop optimizations in terms of performance and power tradeoffs. Both low-level loop optimizations at code generation (back-end) phase, such as *loop unrolling* and *software pipelining*, and high-level loop optimizations at program analysis and transformation phase (front-end), such as *loop permutation* and *tiling*, are studied.

In this study, we use the Delaware Power-Aware Compilation Testbed (Del-PACT) — an integrated framework consisting of a modern industry-strength compiler infrastructure and a state-of-the-art micro-architecture-level power analysis platform. Using Del-PACT, the performance/power tradeoffs of loop optimizations can be studied quantitatively. We have studied such impact on several benchmarks under Del-PACT.

The main observations are:

- Performance improvement (in terms of timing) is correlated positively with energy reduction.
- The impact on energy consumption of high-level and low-level loop optimizations is often closely coupled, and we should not evaluate individual effects in complete isolation. Instead, it is very useful to assess the combined contribution of both, high-level and low-level loop optimizations.
- In particular, results of our experiments are summarized as follow:
  - Loop unrolling reduces execution time through effective exploitation of ILP from different iterations and results in energy reduction.
  - Software pipelining may help in reducing total energy consumption – due to the reduction of the total execution time. However, in the two benchmarks we tested, the effects of high-level loop transformation cannot be ignored. In one benchmark, even with software pipelining disabled, applying proper high-level loop transformation can still improve the overall execution time and energy, comparing with the scenario where high-level loop transformation is disabled though software pipelining is applied.
  - Some high-level loop transformation such as loop permutation, loop tiling and loop fusion contribute significantly to energy

reduction. This behavior can be attributed to reducing both the total execution time and the total main memory activities (due to improved cache locality).

An analysis and discussion of our results is presented in section 4.

1 Introduction

Low power design and optimization [8] are becoming increasingly important in the design and application of modern microprocessors. Excessive power consumption has serious adverse effects – for example, the usefulness of a device or equipment is reduced due to the short battery life time.

In this paper, we focus on compiler optimization as a key area in low-power design [7, 13]. Many traditional compiler optimization techniques are aimed at improving program performance such as reducing the total program execution time. Such performance-oriented optimization may also help to save total energy consumption since a program terminates faster. But, things may not be that simple. For instance, some of such optimization my try to improve performance by exploiting instruction-level parallelism, thus increasing power consumption per unit time. Other optimization may reduce total execution time without increasing power consumption. The tradeoffs of these optimizations remain an interesting research area to be explored.

In this study, we are interested in the impact of loop optimizations in terms of performance and power tradeoffs. Both low-level loop optimizations at code generation (back-end) phase, such as *loop unrolling* and *software pipelining*, and high-level loop optimizations at program analysis and transformation phase (front-end), such as *loop permutation* and *tiling*, are studied.

Since both high-level and low-level optimization are involved in the study, it is critical that we should use a experimental framework where such tradeoff studies can be conducted effectively. We use the Delaware Power-Aware Compilation Testbed (Del-PACT) — an integrated framework consisting of a modern industry-strength compiler infrastructure and a state-of-the-art micro-architecture level power analysis platform. Using Del-PACT, the performance/power tradeoffs of loop optimizations can be studied quantitatively. We have studied the such impact on several benchmarks under Del-PACT.

This paper describes the motivation of loop optimization on program performance/power in Section 2 and describing the Del-PACT platform in Section 3. The results of applying loop optimization on saving energy are given in Section 4. The conclusions are drawn in Section 5.

2 Motivation for Loop Optimization to Save Energy

In this section we use some examples to illustrate the loop optimizations which are useful for energy saving. Both low-level loop optimizations at the code generation (back-end) phase, such as *loop unrolling* and *software pipelining*, and high-level loop optimizations at the program analysis and transformation phase (front-end), such as *loop permutation*, *loop fusion* and *loop tiling*, are discussed.

2.1 Loop unrolling

Loop unrolling [17]intends to increase instruction level parallelism of loop bodies by unrolling the loop body multiple times in order to schedule several loop iterations together. The transformation also reduces the number of times loop control statements are executed.

2.2 Software pipelining

Software pipelining restructures the loop kernel to increase the amount of parallelism in the loop, with the intent of minimizing the time to completion. In the past, resource-constrained software pipelining [10, 16] has been studied extensively by several researchers and a number of modulo scheduling algorithms have been proposed in the literature. A comprehensive survey of this work is provided by Rau and Fisher in [15]. The performance of software pipelined loop is measured by II(*initiation interval*). Every II cycles a new iteration is initiated, thus throughput of the loop is often measured by the value of II derived from the schedule. By reducing program execution time, software pipelining helps reduce the total energy consumption. But, as we will show later in the paper, the net effect of energy consumption due to software pipelining also depends on high-level loop transformations performed earlier in the compilation process.

2.3 Loop permutation

Loop permutation (also called loop interchange for two dimensional loops) is a useful high-level loop transformation for performance optimization [19]. See the following C program fragment:

```
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    a[j][i] = 1;
  }
}
```

Since the array `a` is placed by row-major mode, the above program fragment doesn't have good cache locality because two successive references on array `a` have a large span in memory space. By switching the inner and outer loop, the original loop is transformed into:

```
for (j = 0; j < N; j++) {
  for (i = 0; i < M; i++) {
    a[j][i] = 1;
  }
}
```

Note that the two successive references on array `a` access contiguous memory address thus the program exhibits good data cache locality. It usually improves both the program execution and power consumption of data cache.

2.4 Loop tiling

Loop tiling is a powerful high-level loop optimization technique useful for memory hierarchy optimization [14]. See the matrix multiplication program fragment:

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < N; k++) {
      c[i][j] = c[i][j] +
        a[i][k] * b[k][j];
    }
  }
}
```

Two successive references to the same element of `a` are separated by N multiply-and-sum operations. Two successive references to the same element of `b` are separated by N^2 multiply-and-sum operations. Two successive references to the same element of `c` are separated by 1 multiply–and-sum operation. For the case when N is large, references to `a` and `b` exhibits little locality and the frequent data fetching from memory results in high power consumption.

Tiling the loop will transforme it to:

```
for (i = 0; i < N; i+=T) {
  for (j = 0; j < N; j+=T) {
    for (k = 0; k < N; k+=T) {
      for (ii = i; ii < min(i+T, N); ii++) {
      for (jj = j; jj < min(j+T, N); jj++) {
      for (kk = k; kk < min(k+T, N); kk++) {
```

```
      c[ii][jj] = c[ii][jj] +
        a[ii][kk] * b[kk][jj];
     }
     }
     }
    }
  }
}
```

Notice that in the inner three loop nests, we only compute a partial sum of the resulting matrix. When computing this partial sum, two successive references to the same element of a are separated by T multiply-and-sum operations. Two successive references to the same element of b are separated by T^2 multiply-and-sum operations. Two successive references to the same element of c are separated by one multiply-and-sum operation. A cache miss occurs when the program execution re-enter the inner three loop nests after i, j or k is incremented. However, cache locality in the inner three loops is improved.

Loop tiling may have dual effects in improving total energy consumption: it reduces both the total execution time and the cache miss ratios – both help energy reduction.

2.5 Loop fusion

See the following program fragment:

```
for (i = 0; i < N; i++) {
  a[i] = 1;
}

for (i = 0; i < N; i++) {
  a[i] = a[i] + 1;
}
```

Two successive references to the same element of a span the whole array a in the code above. By fusing the two loops together, we can get the following code fragment:

```
for (i = 0; i < N; i++) {
  a[i] = 1;
  a[i] = a[i] + 1;
}
```

The transformed code has much better cache locality. Just like loop tiling, this transformation will reduce both power and energy consumption.

3 Power and Performance Evaluation Platform

It is clear that, for the purpose of this study, we must use a compiler/simulator platform which (1) is capable of performing loop optimizations at both the high-level and the low-level, and a smooth integration of both; (2) is capable of performing micro-architecture level power simulations with a quantitative power model.

To this end, we chose to use the Del-PACT(Delaware Power-Aware Compilation Testbed) – a fully integrated framework composed of SGI MIPSpro compiler retargeted to the SimpleScalar [1] instruction set architecture, and a micro-architecture level power simulator based on an extension of the SimpleScalar architecture simulator instrumented with the Cai/Lim power model [5, 4], as shown in Figure 1. The SGI MIPSpro compiler is an industry-strength highly optimizing compiler. It implements a broad range of optimizations, including interprocedural analysis and optimization (IPA), loop nest optimization and parallelization (LNO) [18], and SSA-based global optimization (WOPT) [2, 11] at high level. It also has an efficient backend including software pipelining, integrated global and local scheduler(IGLS) [12], and efficient global and register allocators (GRA and LRA) [3]. The legality of loop nest optimizations listed in Section 2 depends on dependence analysis [20]. The SGI MIPSpro compiler performs alias and dependence analysis and a rich set of loop optimizations including those we will study in the paper. We have ported the MIPSpro compiler to the SimpleScalar instruction set architecture.

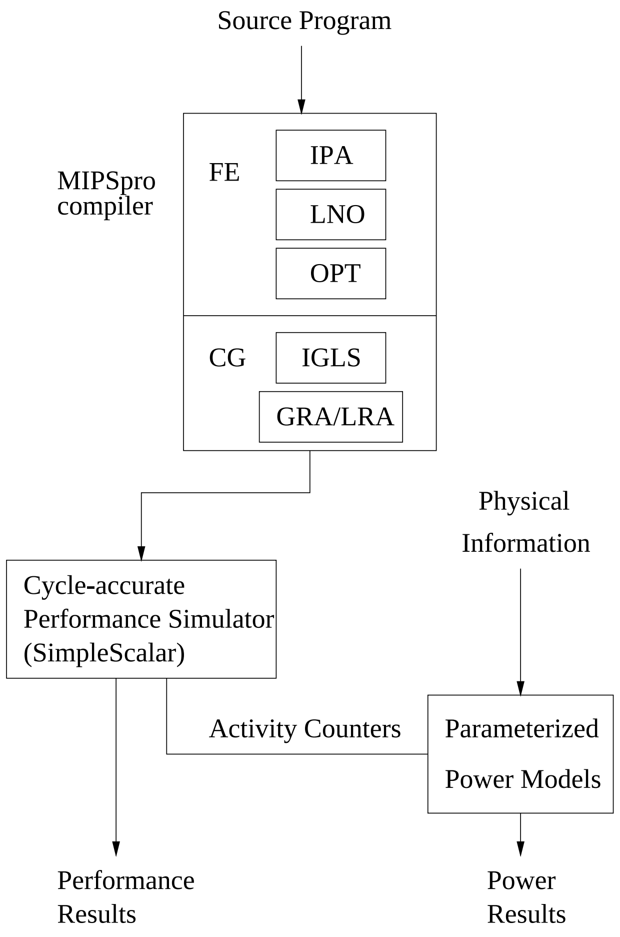

Figure 1: Power and Performance Evaluation Platform

The simulation engine of the Del-PACT platform is driven by the Cai/Lim power model as shown in the same diagram. The instrumented SimpleScalar simulator generates performance results and *activity counters* for each functional block. The physical information comes from approximation of circuit level power behaviors. During each cycle, the parameterized power model computes the present power consumption of each functional unit using the following formula:

$$\text{power} = AF * PDA * A + \text{idle power} + \text{leakage power}$$

AF Activity factor

PDA Active power density

A Area

The power consumption of all functional blocks is summed up, thus obtaining the total power consumption.

Other power/performance evaluation platforms exist as well. A model worth mentioning is SimplePower [9]. In [9] loop transformation techniques are evaluated. In their framework, high-level transformation and low-level loop transformations are performed in two isolated compilers while in our platform these two are tightly coupled into a single compiler. The difference between these two power models are left to be studied and a related work is found in [6].

4 Experimental Results

In this section, we present the experiments we have conducted using Del-PACT platform. Two benchmark programs: *mxm* and *vpenta* from the SPEC92 floating point benchmark suite are used. We evaluated the impact on performance/power of loop nest optimizations, software pipelining and loop unrolling. Loop nest optimization is a set of high-level optimizations that includes loop fusion, loop fission, loop peeling, loop tiling and loop permutation. The MIPSpro compiler analyzes the compiled program by determining the memory access sequence of loops, choosing those loop nest optimizations which are legal and profitable. Looking through the transformed code, we see that the loop nest optimizations applied on *mxm* is loop permutation and loop tiling, while those applied on *vpenta* are loop permutation and loop fusion. Performance, power and energy results of these transformations on each benchmark are shown in Figure 2.

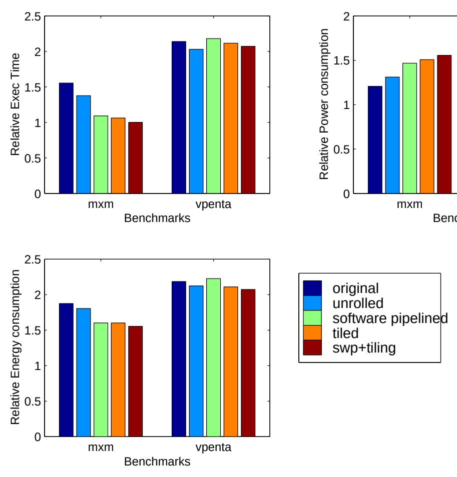

Figure 2: Performance, power and energy comparison

We observe that the performance improvement in terms of timing is correlated positively with the energy reduction. From Figure 2 we see the variation of execution time causes the similar variation in energy consumption. The results show that in the two benchmarks we have run, the dominating factor of energy consumption is the execution time.

Loop unrolling improves the program execution by increasing instructions level parallelism thus increasing power consumption correspondingly. For the *mxm* example, the instructions per cycle(IPC) increased from 1.68 to 1.8 by unrolling 4 times. For the *vpenta* example, loop unrolling reduces the total instruction count by 2% because of cross-iteration common subexpressions elimination. The IPC value before the loop unrolling and after that are 1.01 and 1.04 respectively.

Software pipelining helps in reducing total energy consumption in the *mxm* example. The IPC value without and with software pipelining are 1.68 and 1.9 respectively. Power consumption increase a little bit more as opposed to the case with loop unrolling because software pipelining exploits more instruction level parallelism than loop unrolling does. However, energy consumption is still reduced compared with the original untransformed program. In *vpenta* example, software pipelining does not help because of the high miss rate(13%) of level-1 data cache accesses.

Loop tiling and loop permutation applied on *mm* enhanced cache locality and they can improve the program performance more than software pipelining does. Loop permutation and loop fusion help the *vpenta* program reduce its level-1 data cache miss rate from 13% to 10%, thus reducing total energy consumption. Also these transformations make the performance improvement of software pipelining more evident compared with the case that software pipelining is applied without these high-level optimizations.

5 Conclusions

In this paper, we introduced our Del-PACT platform, which is an integrated framework that includes the MIPSpro compiler, SimpleScalar simulator and CAI/LIM power estimator. This platform can serve as the tool to make architecture design tradeoffs, and to study the impact of compiler optimization on program performance and power consumption. We use this platform to conduct experiments on the impact of loop optimizations on program performance vs power.

References

[1] Todd Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, Univ of Wisconsin, 1997.

[2] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination

based on SSA form. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 273–286, Las Vegas, Nev., Jun. 15–18, 1997. *SIGPLAN Notices*, 32(6), Jun. 1997.

[3] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

[4] A. Dhodapkar, C.H.Lim, and G.Cai. Tempest: A thermal enabled multi-model power/performance estimator. In Workshop on Power-Aware Computer Systems, Nov 2000.

[5] G.Cai and C.H.Lim. Architectural level power/performance optimization and dynamic power estimation. Cool Chips Tutorial, in conjunction with 32nd Annual International Symposium on Microarchitecture. Haifa, Israel, Nov 1999.

[6] Soraya Ghiasi and Dirk Grunwald. A comparison of two architectural power models. In Workshop on Power-Aware Computer Systems. Cambridge, MA, Nov 2000.

[7] Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Low power design methodologies: Hardware and software issues. Tutorial on Parallel Architecture and Compilation Techniques 2000.

[8] J.M.Rabaey and M. Pedram, editors. *Low-Power Design Methodologies*. Kluwer, 1996.

[9] Mahmut T. Kandemir, N. Vijaykrishnan, Mary Jane Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 304–307, NY, June 5–9 2000. ACM/IEEE.

[10] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Geor., Jun. 22–24, 1988. *SIGPLAN Notices*, 23(7), Jul. 1988.

[11] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 26–37, Montréal, Qué., Jun. 17–19, 1998. *SIGPLAN Notices*, 33(6), Jun. 1998.

[12] Srinivas Mantripragada, Suneel Jain, and Jim Dehnert. A new framework for integrated global local scheduling. In *Proc. of the 1998 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 167–174, Paris, Oct. 12–18, 1998. IEEE Comp. Soc. Press.

[13] M.Kandemir, N. Vijaykrishnan, M. J. Irwin, W. Ye, and I. Demirkiran. Register relabeling: A post-compilation technique for energy reduction. In *Workshop on Compilers and Operating Systems for Low Power 2000 (COLP'00)*.

[14] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[15] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *J. of Supercomputing*, 7:9–50, May 1993.

[16] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific

computing. In *Proc. of the 14th Ann. Microprogramming Work.*, pages 183–198, Chatham, Mass., Oct. 12–15, 1981. ACM SIGMICRO and IEEE-CS TC-MICRO.

[17] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 2000 international conference on Supercomputing*, pages 153 – 166, Santa Fe, NM USA, May 8 - 11 2000.

[18] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the 29th Ann. Intl. Symp. on Microarchitecture*, pages 274–286, Paris, Dec. 2–4, 1996. IEEE-CS TC-MICRO and ACM SIGMICRO.

[19] Michael Wolfe. Advanced loop interchanging. In *Proc. of the 1986 Intl. Conf. on Parallel Processing*, pages 536–543, St. Charles, Ill., Aug. 19–22, 1986.

[20] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.