Influence of Compiler Optimizations on System Power*

M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye
Microsystems Design Lab
The Pennsylvania State University
University Park, PA 16802
(kandemir,vijay,mji,wye)@cse.psu.edu

ABSTRACT

High-level compiler optimizations have been widely used to achieve speedups on array-based codes. Such optimizations are becoming increasingly important in embedded signal processing and multimedia systems. The focus of these optimizations has traditionally been on improving performance. However, energy constraints are of critical importance in battery-operated embedded devices. In this paper, we present an experimental evaluation of several state-of-the-art compiler optimizations on energy consumption, considering both the processor core (datapath) and memory system. This is in contrast to many of the previous works that have considered them in isolation.

1. INTRODUCTION

Due to their particular characteristics and specific usage (e.g., mobile computing), embedded systems have stringent energy[1] and area constraints. A growing trend in embedded systems, therefore, is integrating the entire system on a single chip, which is beneficial from power as well as form factor points of view. This high-level of integration, that has been made possible by deep sub-micron processing technology, combined with the desire for fast design cycles suggests the use of programmable components and an increasing amount of software implementation relative to hardware. However, we need a powerful compiler technology in order to take full advantage of software implementation.

Over the last two decades numerous compiler optimizations have been proposed to make user programs automatically run their fastest (see for example [8] and the references therein). Some of these optimizations are high-level in the sense that they are applied at source-level, where program access patterns imposed by loop and other control structures are visible. With the advent of parallel architectures and systems with deep memory hierarchies, these optimizations gained momentum and various additional locality and parallelism optimizations have recently been proposed. Loop nest optimizations, that hold an important place in high-level optimizations, specifically target loop nests where most of the execution time is spent (especially in multidimensional signal processing and video applications). Techniques such as loop permutations, loop tiling, and loop fusion have been proven to be very useful in optimizing performance of loop nests, e.g., enhancing cache performance and/or improving parallelism. While such techniques have been thoroughly evaluated from the performance point of view, there has been little effort to analyze their energy impact [2].

In this paper, we present a quantitative evaluation of the impact of different state-of-the-art high-level compilation techniques on energy consumption. We measure energy consumption using a transition-sensitive, cycle-accurate, RT-level energy simulator [9] for a set of representative codes from the multidimensional array domain. This domain is important for embedded systems as it includes a large portion of video and signal processing applications. We study the energy consumed by the entire system for both the original and compiler-transformed codes. The energy consumption of the core[2] is obtained using our cycle-accurate simulator, while the energy consumed by the buses, cache and memory is obtained from an analytical model. We use the modified form of a source-to-source program translator that performs loop and data optimizations to transform the codes [4]. This study is in contrast with most of the prior work where power consumption on only specific system components (e.g., cache and main memories) was investigated [7]. With the help of our cycle-accurate simulator, the source-to-source translator, and a number of benchmark codes, we study the system-wide power effects of six high-level compiler optimizations and gain some insight into tradeoffs between power and performance.

The rest of this paper is organized as follows. Section 2 introduces the compiler optimizations and discusses their expected impact from the energy point of view. The energy simulator and energy models used in this study are described in Section 3. Section 4 explains the application of these optimizations and presents our experimental results. Finally,

*(This work was sponsored in part by grants from NSF (MIP-9705128), Sun Microsystems, and Intel)

[1]Throughout this paper, we use the terms power and energy interchangeably. However, power savings and energy savings may not necessarily go hand-in-hand. The choice of metric used will depend on the application constraints.

[2]Energy consumed in the core includes only the datapath; the control unit energy is not included.

we provide conclusions in Section 5.

2. COMPILER OPTIMIZATIONS

In this section, we introduce the optimizations evaluated (i.e., linear loop transformations, tiling, unrolling, fusion, fission, and scalar expansion) and discuss their impact on power consumption. While these optimizations can improve the performance characteristics of programs enormously [8], their impact on power depends on the relative power dissipation of different hardware components in the system. Among the various high-level compiler transformations, we choose to target loop optimizations for two reasons. First, the multimedia and signal processing applications operate on multidimensional array structures that benefit from such optimizations. Second, these optimizations are widely used by commercial and academic optimizing compilers.

Linear loop transformations attempt to improve cache performance, instruction scheduling, and iteration-level parallelism by modifying the traversal order of the iteration space of the loop nest. The simplest form of loop transformation, called loop interchange, can improve data locality (cache utilization) by changing the order of the loops. From the power consumption point of view, by applying this transformation we can expect a reduction in the total memory power due to better utilization of the cache [6; 5]. For the power consumed in other parts of the system, we do not anticipate a major variation for this small example after the transformation. This may not be true in general, though, as some loop transformations can result in complex loop bounds and array subscript expressions that can potentially increase the power consumed in the core datapath.

Another important technique used to improve cache performance is *blocking,* or *tiling.* When it is used for cache locality, arrays that are too big to fit in the cache are broken up into smaller pieces (to fit in the cache). When we consider power, potential benefits from tiling depend on the changes in power dissipation induced by the optimization on different system components. We can expect a decrease in power consumed in memory, due to better data reuse [2]. On the other hand, in the tiled code, we traverse the same iteration space of the original code using twice as many loops (in the most general case); this entails extra branch control operations and macro calls. These extra computations might increase the power dissipation in the core.

Loop unrolling reduces the trip count of a given loop by putting more work inside the nest with the aim of reducing the number of memory accesses and promoting the register reuse. From the power point of view, fewer accesses to the memory means less power dissipation. In addition, we can also expect a reduction in the power consumed in the register file and data buses.

Loop fusion combines two loops into one loop. Since it improves data reuse, it can reduce the power consumed in the memory system. And, if used with scalar replacement, it can eliminate a number of memory references again reducing the memory system power. Also, since it reduces the number of loop nests, it eliminates lots of branch instructions that would otherwise contribute to a significant percentage of the core power. In *loop fission,* the compiler breaks down a loop into two (or more) separate loops. When done solely for optimizing the iteration scheduling, this transformation can increase power consumption in memory system and elsewhere due to an increase in the number of loop nests and a decrease in temporal locality. Therefore, the power reductions obtained through improved scheduling should be carefully weighed against the extra power consumed in memory.

Scalar expansion is one of the most effective techniques for the exploitation of parallelism among loop iterations [8]. It helps to improve parallelism for the nest whose iterations cannot be run in parallel because of a global scalar variable shared across many iterations. It replaces the said global scalar variable by private array variables for each iteration, thereby allowing each iteration to execute independently. As far as the power consumption is concerned, such a transformation is expected to increase power in the memory system as well as in the datapath (core). The power increase in the core is due to the newly introduced array subscript calculations. The power increase in the memory system, on the other hand, is because of the extra storage requirement in order to accommodate the expanded variable. Depending on power gains that can be obtained through iteration parallelism [3], the extra power dissipation may or may not be problematic.

3. SIMULATOR AND ENERGY MODELS

SimplePower is an execution-driven, cycle-accurate, RT-level power estimation tool built in-house. It is based on the architecture of a five-stage pipelined datapath. The instruction set architecture is a subset of the instruction set (the integer part) of *SimpleScalar*, which is a suite of publicly available tools to simulate modern microprocessors [1]. The results presented in this paper use the *SimplePower* technology tables configured for 0.8μ technology and 3.3V. This paper only considers the data cache memory hierarchy, using the models proposed in [6]. Further, the *SimplePower* core stall cycles due to cache misses affect the core power negligibly due to efficient power management (using clock gating) during stalls. Hence, we report only a single core power measure for the different cache configurations.

A C source benchmark is compiled by the *SimpleScalar* version of `gcc`, which generates *SimpleScalar* assembly codes. The *SimpleScalar* assembler `gas` and loader/linker `gld` produce *SimplePower* executables that can then be loaded into *SimplePower* main memory and executed by *SimplePower core*. In our study, we enhanced a source-to-source optimizer [4] to perform the various code transformation investigated. The simulator can be configured using the command line to set the caches parameters, output the pipeline trace cycle-by-cycle, and dump the memory image. *SimplePower* provides the register file final status, total number of cycles in execution, number of transitions in on-chip buses, switch capacitance statistics for each pipeline stage, switch capacitance statistics for different functional units, and the total switch capacitance.

4. EXPERIMENTS

In this section, we present our experimental results and make some observations. To evaluate our six optimizations, we used five benchmark codes (all but one are from Spec; see Table 1). The same table also shows the miss rates of

Program	Array Sizes	Miss Rate	Optimizations
adi	100*100*2	0.0979	linear transforms, tiling
hydro2d/fct	52*52	0.0962	loop fusion
nasa7/btrix	100*100*100*5	0.2063	loop fusion
nasa7/cholesky	52*52	0.1109	loop fission
tomcatv	100*100	0.2403	scalar expansion

Table 1: Programs used in the experiments.

the original code versions for a 1K, direct-mapped cache and gives the optimizations that we used for each code.

How the source-to-source translator applies optimizations for each code is important as there may be more than one way of applying a given optimization for a given code. Each has different performance and energy implications.

In `adi`, the linear loop optimizations interchanged the order of two loops in the main nest in an attempt to obtain stride-one accesses in the innermost loop, thereby improving spatial locality. (The original version is denoted by `orig` and the tiled version is denoted by `tile`). When we enable tiling, the compiler tiled the innermost loop and hoisted the tile loop (i.e., the loop that iterates on tiles) to the outermost position. Note that in order to run this code standalone, we added a two-deep initialization nest (i.e., a nest that contains two nested loops). The linear transformation permuted this nest whereas tiling did not modify it due to its relatively small contribution to the overall performance.

In `nasa7/btrix`, in order to isolate the impact of loop fusion, we disabled other loop optimizations, and experimented with only original (`orig`) and fused (`fuss`) versions. When fusion is activated, the compiler fused two large one-dimensional (one-deep) loop nests into a very large loop. This example gives us the opportunity for observing the impact of loop fusion (in its extreme, when the resulting loop body gets very large and the chances for intra- and inter-array conflict misses in the data cache increase greatly) on power dissipation. In `hydro2d/fct`, we again measured the impact of fusion on power consumption using the original (`orig`) and the fused (`fuss`) versions. This time the compiler fused both initialization nests (three of them) as well as two main loop nests (each two-deep). In comparison to `nasa7/btrix`, the resulting loop bodies are not very large.

In optimizing `tomcatv`, the compiler applied scalar expansion to the largest loop in the code in an attempt to improve iteration-level parallelism. As a result, 13 scalars are converted to one-dimensional arrays, enabling better loop scheduling. We refer to this version as `expn`. For a smaller nest in the same code, however, the compiler chose not to apply scalar expansion, as it estimated that the performance overhead will offset any potential gain that would be obtained from improved loop scheduling. Finally, in `nasa7/cholesky`, the compiler, when enabled, applied loop fission to a large loop nest in the code. In doing so, it tries to achieve two things: isolating disjoint data dependence cycles by putting them in separate nests, and enabling other transformations, in particular loop permutations, to improve locality further. The loop distributed version is referred to as `fiss`.

	Core Energy (J)	Memory Energy (J)				
		↓→	1-way	2-way	4-way	8-way
orig	0.0043	1K	0.1604	0.0915	0.0794	0.0772
		2K	0.1159	0.0789	0.0756	0.0757
		4K	0.1000	0.0763	0.0759	0.0760
		8K	0.0730	0.0681	0.0742	0.0766
loop	0.0054	1K	0.1418	0.0630	0.0526	0.0468
		2K	0.0844	0.0493	0.0435	0.0436
		4K	0.0609	0.0441	0.0439	0.0440
		8K	0.0378	0.0283	0.0231	0.0251
tile	0.0052	1K	0.1404	0.0731	0.0729	0.0688
		2K	0.0942	0.0646	0.0674	0.0689
		4K	0.0550	0.0426	0.0465	0.0457
		8K	0.0345	0.0228	0.0220	0.0221

Table 2: Energy consumption in `adi`.

	Core Energy (J)	Memory Energy (J)				
		↓→	1-way	2-way	4-way	8-way
orig	0.1565	1K	8.4840	3.9372	2.8179	2.9734
		2K	3.3221	2.3311	1.3897	1.2614
		4K	1.6816	1.3939	1.1123	1.1155
		8K	1.3291	0.9573	0.8942	0.8752
fuss	0.1748	1K	9.4086	4.4788	3.1901	3.2580
		2K	3.9100	2.5048	1.4469	1.2732
		4K	1.8087	1.4712	1.1003	1.1033
		8K	1.3887	0.9480	0.8852	0.8652

Table 3: Energy consumption in `nasa7/btrix`.

	Core Energy (J)	Memory Energy (J)				
		↓→	1-way	2-way	4-way	8-way
orig	0.0008	1K	0.0290	0.0117	0.0079	0.0079
		2K	0.0130	0.0069	0.0060	0.0054
		4K	0.0086	0.0055	0.0054	0.0054
		8K	0.0066	0.0055	0.0055	0.0053
fuss	0.0006	1K	0.0277	0.0102	0.0073	0.0068
		2K	0.0095	0.0074	0.0061	0.0063
		4K	0.0069	0.0050	0.0050	0.0050
		8K	0.0057	0.0050	0.0050	0.0050

Table 4: Energy consumption in `hydro2d/fct`.

	Core Energy (J)	Memory Energy (J)				
		↓→	1-way	2-way	4-way	8-way
orig	0.2717	1K	10.0226	8.4229	7.4468	6.8649
		2K	9.0886	7.9532	7.0437	6.8011
		4K	7.5097	7.1870	6.3485	6.2203
		8K	5.4660	5.4774	5.5711	5.5755
fiss	0.2875	1K	11.0594	10.1304	9.8596	9.8654
		2K	9.4693	9.4344	9.6597	9.7468
		4K	7.4571	6.0289	6.1112	5.8381
		8K	5.2994	5.2730	5.1896	5.2517

Table 5: Energy consumption in `nasa7/cholesky`.

	Core Energy (J)	Memory Energy (J)				
		↓→	1-way	2-way	4-way	8-way
orig	0.0222	1K	1.6934	0.7539	0.5858	0.5731
		2K	0.8384	0.6609	0.5706	0.5707
		4K	0.7920	0.5813	0.5716	0.5720
		8K	0.7111	0.5260	0.5301	0.5296
expn	0.0578	1K	1.4255	0.9555	0.9278	0.9233
		2K	1.1733	0.9347	0.9223	0.9232
		4K	1.0298	0.9162	0.9253	0.9262
		8K	0.9324	0.8677	0.8905	0.8875

Table 6: Energy consumption in `tomcatv`.

We conducted experiments with different cache sizes (1K, 2K, 4K, and 8K) and different set associativities (direct-mapped (1-way), 2-way, 4-way, and 8-way) for each code. All the reported energy values are in Joules (J). Also, the bus power is included in the memory system power in all the results. We did not go beyond 8K because the 1K - 8K range is more realistic for the embedded computing domain. Also, since the cache miss and hit rates for 4-way and 8-way set associative caches were very similar, we do not report results with larger associativities.

In the `adi` code, the number of memory accesses per computation is very high. This is because the it accesses three-dimensional arrays using two-deep loop nests. Consequently, as shown in Table 2, the core power is very low compared to memory power for all cache configurations. An optimizing compiler can be very aggressive in applying potential optimizations such as tiling, if it can detect that the number of memory references per computation is very high. However, we note that applying tiling increases the core energy consumption.

In the next two examples, we evaluated the impact of loop fusion on core and memory system energy consumption. In `nasa7/btrix`, the loop fusion interferes with loop scheduling as the loop body becomes very large. This increases the core power (by 12%) as well as memory power (due to poor scheduling of memory operations) as shown in Table 3. Unfortunately, scalar replacement could not eliminate the large number of memory references. On the other hand, if the cache size is large or the associativity is high, scheduling memory operations becomes less critical (unless the cache is direct-mapped, in which case poor scheduling induces more conflict misses). For example, for a 4K, 4-way set associative cache, the fused version is marginally better than the original. In the `hydro2d/fct` case, the same optimization is more successful from the energy point of view (see Table 4). It reduces the core power as well as the memory system power by as much as 25%. The reduction in the memory power comes from reductions in the number of memory references rather than from hit/miss rate variations. Overall, we believe that, if applied judiciously, the loop fusion can reduce both the core and memory system power.

We quantified the impact of loop fission (loop distribution) on energy using the `nasa7/cholesky` benchmark (see Table 5). As expected, in general, it increased the core as well as the memory system energy consumption. The increase in the core power is due to the increase in number of control statements as a result of loop fission (as it increases the number of nests in the program). The increase in the memory power, on the other hand, is due to an increased number of memory references. In large caches (4K and above), however, the negative impact of this optimization disappears.

Finally, we used `tomcatv` to investigate the energy performance of scalar expansion. The results, given in Table 6, indicate that this optimization (like loop fission) increases the core as well as the memory system power. The increase in the core power (by a factor of 2.6 times) is due to the extra array subscript expression calculations and the increase in the memory system power is due to an increase in the number of memory accesses. This negative impact can be offset, depending on the energy saving that is to be obtained through better parallelism, if the instruction set architecture supports it. Given a parallelization strategy, architecture and a suitable power estimation model, an optimizing compiler can quantify such tradeoffs for regular codes.

5. CONCLUSIONS

Power minimization represents an important aspect of embedded system design. Increasing employment of software in these systems makes it necessary to evaluate the impact of frequently used optimizations on system energy consumption. This paper is a step in this direction. Through extensive simulation, we investigated the effect of six state-of-the-art compiler optimizations on several benchmark codes from the multidimensional array domain. Our results indicate that the energy consumed in the memory system is higher than the core in unoptimized codes. We have also observed that, in contrast to reducing the memory system energy, most optimizations (except loop unrolling) increase the energy consumed in the core. Since these compiler optimizations can be useful for embedded video and signal processing systems, we recommend that system designers continue their efforts to reduce the core power.

6. REFERENCES

[1] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. *Technical Report CS-TR-96-103,* Computer Science Dept., University of Wisconsin, Madison, July 1996.

[2] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In Proc. *the IEEE Workshop on VLSI Signal Processing,* pages 178-187, 1994.

[3] A. Chandrakasan et.al., Optimizing power using transformations, *IEEE Transactions on CAD*, TCAD-14(1):12-31, Jan. 1995.

[4] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In Proc. *MICRO-31,* Dallas, TX, December, 1998

[5] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In Proc. *ISSS'97,* Antwerp, Sept 1997.

[6] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In Proc. *DAC'99,* New Orleans, Louisiana, 1999.

[7] C. Su and A. Despain. Cache design trade-offs for power and performance optimization: a case study. In Proc. *International Symposium on Low Power Electronics and Design,* pp. 63–68, 1995.

[8] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

[9] W. Ye, N. Vijaykrishnan, M. Kandemir and M. J. Irwin. *The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool*, In Proc. *DAC'00,* Los Angeles, CA, 2000.