Chapter 10

COMPILER OPTIMIZATIONS FOR LOW POWER SYSTEMS

Mahmut Kandemir, N. Vijaykrishnan, Mary Jane Irwin
Microsystems Design Lab
Pennsylvania State University
*University Park, PA**

Abstract Most current compiler optimizations focus on improving execution time. With the increasingly widespread use of embedded systems, however, power/energy consumption is also becoming an important issue. This is particularly true for battery-operated devices where power consumption has first class status along with performance and form factor.

This paper makes the following contributions. First, we present two low-level (back-end) compiler optimizations for energy reduction. An important conclusion drawn from evaluating the impact of these optimizations is that compiling for power/energy is different from compiling for execution cycles. Second, we evaluate widely used state-of-the-art high-level compiler optimizations from a power consumption perspective. Further, we compare the relative impact of these high-level optimizations on both energy and performance metrics in order to identify any differences in optimizing for energy and performance. Finally, we cover a set of optimizations that are designed specifically for exploiting low power features supported by the hardware. In particular, we show how loop and data transformation can be used to exploit low-power mode control mechanisms available in some memory architectures.

Keywords: Compiler Optimizations, Low Power Modes, Energy Optimization, Loop and Data Transformations

*Partial funding provided by a grant from GSRC and NSF CAREER awards 0093082 and 0093085 and NSF grant 0073419

1. Introduction

Energy has become an important design consideration, together with performance, in computer systems. While energy conscious design is obviously crucial for battery driven mobile and embedded systems, it has also become important for desktops and servers due to packaging and cooling requirements where power consumption has grown from a few watts per chip to over a 100 watts. As a result, there has been a great deal of interest recently in examining optimizations for energy reduction from both the hardware and software points of view.

While hardware optimizations has been the focus of several studies and are fairly mature, software approaches to optimizing power are relatively new. Progress in understanding the impact of traditional compiler optimizations and developing new power-aware compiler optimizations are important to overall system energy optimization. Software has a significant impact on the overall energy consumption being the main determinant for activity on the processor core, interconnect and memory system, which are, collectively, responsible for significant percentage of total power dissipation. Despite this observation, to date, most of the compiler techniques consider only delay and area as their main performance metrics. With the growing demand for power-aware software, there is an acute need for investigating energy-oriented compilation techniques and their interaction and integration with performance-oriented compiler optimizations. In this chapter, we seek answers for the following questions:

- Is the most efficient code from the performance perspective the same as that for the energy viewpoint? If not, why?
- What is the impact of current performance-oriented software optimizations (that primarily aim at maximizing data locality and enhancing parallelism [16]) on energy? How do they affect the energy consumption of different system components (memory system, datapath, etc.)?
- What are the relative gains obtained using software and hardware optimization techniques? How can one exploit the interaction between these optimizations to reduce energy further?

We believe that any progress made in answering them will pave the way for our understanding of impacts and interactions of hardware and software optimizations.

The rest of this chapter is organized as follows. In Section 2, we present two energy-aware low level compiler optimizations. Next, we evaluate the influence of performance-oriented software optimizations on energy in Section 3. In Section 4, we show how hardware and software optimizations for energy interact with each other. Finally, we conclude in Section 5.

2. Energy-Aware Low-Level Compiler Optimizations

In this section, we present two low-level compiler techniques to reduce energy consumption. First, we show how instruction scheduling can be performed to reduce energy consumption in the datapath. Second, we present a post-compilation optimization that relabels registers to minimize switching energy in the buses.

2.1 Influence of Instruction Scheduling on Energy

Traditionally, optimizing compilers use instruction scheduling or reordering instructions to improve datapath performance. There have been several efforts at tuning the instruction schedulers to optimize energy consumption. A key observation behind all these optimizations is that the major determinant of the energy consumption is the switching activity in the datapath. Thus, by ordering instructions in the schedule such as to reduce this switching activity between successive instructions, an energy-aware schedule can be created. An instruction scheduling algorithm called *cold scheduling* that prioritizes the selection of each instruction based on the energy cost of placing that instruction next into the schedule is proposed by Su et al. [12]. Tiwari et al. [13, 14, 8] and Russell and Jacome [11] used instruction-level energy characterization based on current measurements to study the impact of instruction ordering on energy consumption.

In order to capture, the essence of the energy-oriented scheduling mechanisms, we explain a pure energy-oriented version of list scheduling given in Figure 10.1. Each instruction is assumed to have a base cost, denoting its average energy consumption. A weighted edge between nodes `i` and `j` gives the energy consumed by the activity of switching from instruction `i` to instruction `j`.[1] This edge weight is also called *circuit-state effect* (*circuit-state cost* or *inter-instruction cost*) [13]. Since the type and number of instructions (a process called *instruction selection*) are fixed prior to instruction scheduling, a successful scheduling algorithm can only reduce the total circuit-state effect. The pure energy-oriented scheduling accomplishes this as follows. First, it selects one of the schedulable nodes (say `i`) and schedules it. In the next step, it attempts to select a node `j` such that the circuit state effect between `i` and `j` is minimal among all possible alternatives; in the following step, a node `k` is selected such that the circuit-state cost between `j` and `k` is minimum and so on. This approach is also greedy, in that it tries to minimize the cumulative circuit-state cost up to the current point in each step. One noticeable difference between this approach and the classical performance-oriented list scheduling is that the latter considers the maximum delay among the candidate nodes (that

```
energy_{i,j} – inter-instruction (circuit-state) cost between
          the instructions i and j
last – last instruction executed

  schld=∅
  while can-sch ≠ ∅ do
    if schld = ∅ then
      n = can-sch[1]
    else
      min-energy = + ∞
      for each j ∈ can-sch do
        if energy_{last,j} < min-energy then
          min-energy = energy_{last,j}
          n = j
    last = n
    schld ⊕ = [n]
    can-sch - = {n}
    for each i ∈ succ(n) do
      if ∀m ∈ pred(i) ∃j such that schld[j]=m then
        can-sch ∪ = {i}
  return (schld)
```

Figure 10.1. Energy-Oriented Scheduling

is, it considers the nodes that have not been scheduled yet) whereas the former considers the total circuit-state cost so far (that is, it considers the nodes that have already been scheduled). The algorithms proposed by [12] and [13] also use a similar approach.

Through extensive experimentation presented in [9], we found that a pure performance-oriented scheduling technique does not necessarily generate the most energy-efficient code. A pure energy-oriented scheduling, on the other hand, is found to be quite effective in reducing the energy consumption; additionally, it was only 6% worse than the pure performance scheduling as far as the execution cycles are concerned for studied configurations. Instead of optimizing for energy and performance separately, instruction scheduling techniques that consider energy and delay simultaneously in a unified setting can also be designed. Parikh et. al. show that such combined metrics are effective in reducing both datapath energy and performance.

One of the difficulties of instruction scheduling for energy is in estimating the impact of the data portions of the instructions. Thus, in addition to instruction scheduling, techniques such operand re-ordering (e.g., swapping operands in ALU or floating point operations) can be very effective in reducing switching activity on buses. In the next subsection, we show how to reduce switching activity in the operand fields of instructions.

2.2 Influence of Register Assignment on Bus Energy

In this section, we focus on reducing the switching activity on the Icache data bus (between the processor core and Icache) by relabeling the register fields of the compiler generated instructions. Sample traces are used to record the transition frequencies between register labels (encodings) in the instructions executed in consecutive cycles using *SimplePower* [15]. This information is then used to obtain new encodings for the registers such that the switching activity (and consequently the energy consumption) in the Icache data bus is reduced. The technique is applicable to any system where switching activity imposed by register labels has an impact on overall energy consumption. The encoding of instructions using relabeling can be considered similar to data encoding techniques investigated for data.

To illustrate the idea, let us consider two consecutive `add` instructions in the *SimplePower* assembly language:

`add` s_i, s_j, s_k
`add` s_l, s_m, s_n

In this simple sequence, there are three switching activities between registers: (1) from s_i to s_l in the destination-register slot, (2) from s_j to s_m in the first source-register slot, and (3) from s_k to s_n in the second source-register slot. Depending on the encodings of the registers involved, the impact of these switching activities can be quite significant. These *register transitions* can also occur between different types of instructions.

It should be clear that for register fields that have frequent transitions, we need to use register numbers whose Hamming distance is minimum. The problem is that register assignments are done by the compiler using sophisticated algorithms and considering a number of other important issues such as minimizing register spills and maximizing register reuse. Therefore, we cannot arbitrarily change the register numbers, just to minimize power consumption. Such modifications, among other things, can also violate data dependences across instructions, thereby changing the semantics of the program being optimized. On the other hand, other alternatives, namely, determining a near-optimal register assignment considering both power and performance is very difficult.

We developed a post-pass, polynomial-time algorithm that relabels the registers *after* global register allocation performed by the compiler back-end. Relabeling registers is always legal as long as it is performed throughout the code. For example, if we decide to relabel s_i as s_j, *all* the occurrences of s_i should be changed to s_j. Also, if we are to perform relabeling for multiple pairs, this should be done simultaneously for all pairs. Informally, our post-pass algorithm takes a compiler-generated register assignment (register allocation) as

input and generates an alternative assignment that reduces the power, maintaining the same performance as the original assignment.

The register relabeling optimization was incorporated in the *SimplePower* compilation framework by modifying the Simplescalar toolset. When the optimization was evaluated using several applications, we observed a 12% reduction in the total energy reduction in the Icache data bus using the register relabeling optimization [18].

3. Influence of High-Level Loop Optimizations on Energy

Software techniques aimed at memory power optimizations are more recent as the traditional emphasis has been on improving performance. The goals of these transformations are to reduce the redundancy in data transfers, and to introduce more locality in the accesses so that more data can be retained in registers local to the datapaths and the part of the memory hierarchy closer to the processor. Typically, accessing memory smaller and closer to the datapath reduces the effective capacitance that is switched, thereby reducing the overall energy consumption. Some of the optimizations may, however, result in energy consumption to increase in other system components. In order to understand these tradeoffs better, we evaluate the impact of three widely used high-level compiler optimizations on a simple matrix multiply code.

3.1 Overview of High-Level Optimizations

In this subsection, we provide an overview of the optimizations considered. They are as follows:

Linear Loop Transformations: The 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 [16], 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]. The power consumed in other parts of the system can potentially increase as some loop transformations can result in complex loop bounds and array subscript expressions. This may occur, for example, with loop transformations used for eliminating data dependencies alone.

Loop Tiling: Another important technique used to improve cache performance is blocking, or tiling [17]. 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 depend on the changes in power dissipation induced by the optimization on different system compo-

nents. We can expect a decrease in power consumed in memory, due to better data reuse [10, 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 components of the system. The exact increase/reduction in power consumed in this example will depend on the problem size (`n`), the tile size (`t`), and the architecture under consideration. Note that improving the energy consumption in the memory system using compiler optimizations may make the energies consumed in the core and the memory system comparable, rendering the core power consumption more significant than before.

Loop Unrolling: This optimization unrolls a given loop, thereby reducing loop overhead and increasing the amount of computation per iteration. From the power point of view, fewer computations means less power dissipation. In addition, we can also expect a reduction in the power consumed in the register file and data buses. The power consumed in the datapath will also be affected by the low-level optimizations performed by the back-end compiler.

3.2 Experimental Evaluation

We evaluated the energy consumptions for the matrix multiply code for different cache topologies (configurations) and program versions (each corresponding to different combinations of three optimizations mentioned above). The first observation we made is that all optimizations except loop unrolling increase the core power. This is due to the fact that the optimized versions generally have more complex loop structures; that, in turn, means extra branches and more complex subscript and loop bound calculations. Loop unrolling is an exception, as it reduces loop control overhead and enables better loop scheduling.

When considering the memory power, on the other hand, we made the following observations. First, with the increasing cache size and/or associativity, tiling performs better than pure linear loop transformations and unrolling. Unlike those optimizations, tiling exploits locality in all loop nest dimensions; increasing associativity helps to eliminate conflict misses between different array tiles. Second, in the original (unoptimized) code, the memory power is 5 to 47 times larger than the core power. However, after some optimizations, this picture changes. In particular, beyond a 2K, 2-way set associative cache (i.e., higher associativities or larger caches), the core and memory powers become comparable when some optimizations are applied. For example, when tiling is applied for a 2K, 4-way associative cache, the memory energy is 0.0764 J, which is smaller than the core energy, 0.0837 J. Similarly, for the most optimized version (that uses all three optimizations), the core and memory energy

Version	$\downarrow^{\rightarrow}$	Miss Rates			
		1-way	2-way	4-way	8-way
original	1K	0.1117	0.1020	0.1013	0.1013
	2K	0.0918	0.0989	0.1013	0.1013
	4K	0.0737	0.0330	0.0245	0.0150
	8K	0.0680	0.0214	0.0117	0.0117
linear transformed	1K	0.0278	0.0119	0.0113	0.0104
	2K	0.0185	0.0107	0.0099	0.0099
	4K	0.0135	0.0100	0.0099	0.0099
	8K	0.0118	0.0099	0.0099	0.0099
unrolled	1K	0.0678	0.0384	0.0359	0.0359
	2K	0.0479	0.0362	0.0359	0.0359
	4K	0.0358	0.0198	0.0145	0.0173
	8K	0.0294	0.0135	0.0077	0.0077
tiled	1K	0.0180	0.0055	0.0039	0.0039
	2K	0.0105	0.0028	0.0016	0.0016
	4K	0.0046	0.0016	0.0012	0.0013
	8K	0.0027	0.0008	0.0007	0.0006

Figure 10.2. Miss Rates for the Matrix Multiply Code

consumptions are very close for a 4K, 4-way set associative cache. This shows that when we apply optimizations, we reduce the memory energy significantly making the contribution of the core energy more important. Since we expect these optimizations (in particular, loop tiling) to be applied frequently by optimizing compilers, reducing core power using additional techniques might become very important. Overall, the power optimizations should not focus only on memory, but need to consider the overall system power. In fact, the choice of best optimization for this example depends strongly on the underlying cache topology. For instance, when we consider the total energy consumed in the system, for a 4K, 2-way cache, the version that uses only loop permutation and unrolling performs best. Whereas for an 8K, 8-way cache, the most optimized version (that uses all three optimizations) outperforms the rest. In fact, given a search domain for optimizations and a target cache topology, an optimizing compiler can decide which optimizations will be most suitable.

3.3 Cache Miss Rates versus Energy Consumptions

We now investigate the correlation between cache miss rate and energy consumption. Figure 10.2 gives the miss rates for some selected cases. This subsection will make some correlations between miss rates and energy consumptions. Let us first consider the miss rates and energy consumption of the

original (unoptimized) code. When we move from one cache configuration to another, we have a similar reduction rate for energy as that for miss rate. For instance, going from 1K, 1-way to 1K, 2-way reduces the miss rate by a factor of 1.10 and reduces the energy by the same factor. As another example, when we move from 1K, 1-way to 4K, 8-way, we reduce the miss rate by a factor of 7.45, and the corresponding energy reduction is a factor of 7.20. These results show that the gain in energy obtained by increasing associativity is not offset, in general, by the increasing complexity of the cache topology. As long as a larger or higher-associative cache reduces miss rates significantly (for a given code), we might prefer it, as the negative impact of the additional complexity is not excessive. However, we note that when moving from one cache configuration to another, if there is not a significant change in miss rate (as was the case in our experiments when going from 1K, 4-way to 1K, 8-way), we incur an energy increase. This can be expected as, everything else being equal, a more complex cache consumes more power (due to more complex matching logic).

Next, we investigate the impact of various optimizations for a fixed cache (and memory) topology. The following three measures are used to capture the correlation between the miss rates and energy consumption of the original and optimized versions.

$$\texttt{Improvement}_\texttt{m} = \frac{\text{Miss rate of the original code}}{\text{Miss rate of the optimized code}},$$

$$\texttt{Improvement}_\texttt{e} = \frac{\text{Memory energy consumption of the original code}}{\text{Memory energy consumption of the optimized code}},$$

$$\texttt{Improvement}_\texttt{t} = \frac{\text{Total energy consumption of the original code}}{\text{Total energy consumption of the optimized code}}.$$

In the following discussion, we consider four different cache configurations: 1K, 1-way; 2K, 4-way; 4K, 2-way; and 8K, 8-way. Given a cache configuration, the following table shows how these three measures vary when we move from the original (unoptimized) version to an optimized (tiled) version of the matrix multiply code.

	1K, 1-way	2K, 4-way	4K, 2-way	8K, 8-way
$\texttt{Improvement}_\texttt{m}$	6.21	63.31	20.63	19.50
$\texttt{Improvement}_\texttt{e}$	2.13	18.77	5.75	2.88
$\texttt{Improvement}_\texttt{t}$	1.96	9.27	3.08	1.47

We see that in spite of very large reductions in miss rates as a result of tiling, the reduction in energy consumption is not as high. Nevertheless, it still follows the miss rate. We made the same observation in different benchmark codes as well. We have found that $\texttt{Improvement}_\texttt{e}$ is smaller than $\texttt{Improvement}_\texttt{m}$ by a factor of 2 - 15. Including the core (datapath) power makes the situation worse for tiling (from the energy point of view), as this optimization increases

the core energy consumption. Therefore, compiler writers for energy-aware systems can expect an overall energy reduction as a result of tiling, but not as much as the reduction in the miss rate. Thus, optimizing compilers that estimate the miss rate (before and after tiling) statically at compile time can also be used to estimate an approximate value for the energy variation. The following table gives the same improvement measures for the loop unrolled version of the matrix multiply code.

	1K, 1-way	2K, 4-way	4K, 2-way	8K, 8-way
$\texttt{Improvement}_m$	1.65	2.82	1.67	1.52
$\texttt{Improvement}_e$	2.07	3.53	2.07	1.83
$\texttt{Improvement}_t$	2.03	3.37	1.97	1.68

The overall picture here is totally different. First, $\texttt{Improvement}_e$ is larger than $\texttt{Improvement}_m$, which proves that loop unrolling is a very useful transformation from the energy point of view. Including the core power makes only a small difference, as this optimization reduces the core power as well. We should mention that our other experiments (not presented here due to lack of space) yielded similar results. We now look at the loop transformed version of the same code:

	1K, 1-way	2K, 4-way	4K, 2-way	8K, 8-way
$\texttt{Improvement}_m$	4.02	10.23	3.30	1.18
$\texttt{Improvement}_e$	3.42	8.51	2.74	0.99
$\texttt{Improvement}_t$	3.17	6.84	2.32	0.94

Here, $\texttt{Improvement}_e$ closely follows $\texttt{Improvement}_m$. Including the core energy brings the energy improvement down further, as in this example, the loop optimization results in extra operations for the core. In the experiments with other cache configurations, we observed similar trends: $\texttt{Improvement}_e$ generally follows $\texttt{Improvement}_m$; but it is slightly lower. And, $\texttt{Improvement}_t$ is smaller than $\texttt{Improvement}_e$ by a factor of 1.05 to 1.80.

We can conclude that the energy variations do not necessarily follow miss rate variations in the optimized array-dominated codes.

4. Interaction of Hardware and Software Optimizations

In this section, we focus specifically on memory system energy due to data accesses and illustrate how software and hardware optimizations affect this energy.

4.1 Hardware Optimizations

A host of hardware optimizations have been proposed to reduce the energy consumption. In this section, we focus on two cache optimizations, namely,

block buffering and cache subbanking [5]. Note that none of these optimizations cause a noticeable negative impact on performance. In the block buffering scheme, the previously accessed cache line is buffered for subsequent accesses. If the data within the same cache line is accessed on the next data request, only the buffer needs to be accessed. This avoids the unnecessary and more energy consuming access to the entire cache data and tag array. Multiple block buffers can be thought of as a small sized Level 0 cache. In the cache subbanking optimization, the data array of the cache is divided into several subbanks and only the subbank where the desired data is located is accessed. This optimization reduces the per access energy consumption.

We studied the energy consumed by the matrix multiply code in the data cache with different configurations of block buffers and subbanks (the number of block buffers being either 2, 4 or 8 and the number of sub-banks varying from 1 to 4) for a 4K cache with various associativities. This result showed that increasing the number of sub-banks from one to two provides an energy saving of 45% for the data cache accesses. An additional 22% saving is obtained by increasing the number of sub-banks to 4. It must be observed that the savings are not linear as one may expect. This is because the energy cost of the tag arrays remains constant, while there being a small increase in energy due to additional sub-bank decoding. We found that for block buffering adding a single block buffer reduced the energy by up to 50%. This reduction is achieved by capturing the locality of the buffered cache line, thereby avoiding accesses to the entire data array. However, access patterns in many applications can be regular and repeating across a varied number of different cache blocks. In order to capture this effect, we varied the number of block buffers to two, four, and eight as well. We observed that, for our matrix multiply benchmark, an additional 17% (as compared to a single buffer) energy saving can be achieved using four buffers.

We also found that using a combination of eight block buffers and four subbanks, the energy consumed in 4K (16K) data cache could be reduced on an average by 88% (89%). Thus, such hardware techniques can reduce the energy consumed by processors with on-chip caches. However, if we consider the entire memory system including the off-chip memory energy consumption, the energy savings from these techniques amount to only 4% (15%) when using a 4K (16K) data cache. Thus, it may be necessary to investigate optimizations at the software level to supplement these optimizations.

4.2 Combined Optimizations for Memory Energy

It was found that when a combination of different software (loop tiling, loop unrolling, and linear loop transformations) and hardware (block buffering and

subbanking) optimizations is applied, tiling performs the best among the three individual compiler optimizations applied in terms of memory system energy across different cache configurations. Since, we mentioned earlier that tiling increases the cache energy consumption, subbanking and block buffering are of particular importance here. For the tiled code, moving from a base data cache configuration to one with eight block buffers and four subbanks reduces the overall memory system energy by around 10%. Thus, it is important to use a combination of hardware and software optimizations in designing an energy-efficient system.

Further, we observed that the linear loop transformed codes exploited the block buffers better than the original code and other optimizations. For example, when using two (eight) block buffers in a 4K 2-way cache, the block buffer hit rate was 69% (82%) as compared to the 55% (72%) for the unoptimized matrix multiply code. Thus, it is also important to choose the software optimizations such that they provide the maximum benefits from the available hardware optimizations.

Overall, we observe that even performance based compiler optimizations provide a significantly higher energy savings as opposed to those gained using the pure hardware optimizations considered. However, a closer observation reveals that hardware optimization become more critical for on-chip cache energy reduction when executing optimized codes. We refer the reader to [5] for more discussion on this topic. In the next two sections, we show how we could design software optimizations to improve the effectiveness of low power hardware features.

4.3 Improving Effectiveness of Power Mode Control Mechanisms Using Code Transformations

Memory (DRAM) modules can be placed in different power modes characterized by different energy consumption per cycle and different latencies to service requests. While it is possible to exploit these power modes in optimizing the DRAM energy without modifying the original code (except for inserting instructions to set operating modes), it is also possible to obtain further energy savings by employing computation (loop nest-based) transformations [4]. In this section, we show how loop fission, a widely-used high-level optimization technique can be used to improve the effectiveness of power mode control.

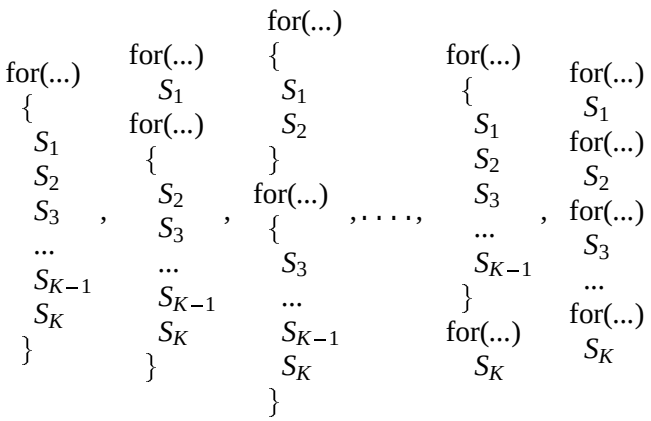
Loop fission (also known as loop distribution [16]) takes a nested loop that contains multiple statements in it, and creates multiple nested loops each with a subset of the original statements. An optimizing compiler can use loop fission for a number of reasons which include improving instruction cache locality and enhancing iteration-level parallelism. This optimization first builds a

statement-level dependence graph (in which the nodes denote statements and the edges correspond to data dependences between them) for the body of the nested loop. Then, if there are no cycles in the dependence graph, the optimization creates a separate loop for each statement. Otherwise, the statements in a (dependence) cycle remain in the same loop. If the nested loop contains multiple loops, loop fission is applied starting from the innermost position, and in fissioning the outer loops, the inner loops are treated as single block statements. Note that even in cases where each statement can be put into a separate loop nest an optimizing compiler may choose not to do so for some other reason such as improving data cache locality or reducing code expansion.

Loop fission helps to improve the effectiveness of array allocation by allowing a finer-granular control over the allocation of arrays. For instance, in the example shown below (assuming that the arrays are of the same size and each memory bank can hold at most two arrays), with the original nest, the two banks that contain the four arrays should be in the active mode throughout the entire execution. After the loop fission, on the other hand, only a single bank needs to be in the active mode during the execution of each loop (assuming that array allocation places `a` and `b` into one bank, and `c` and `d` into the other). The other bank can be put into a low-power mode, thereby saving energy.

```
for(i=0;i<N;i++)                    for(i=0;i<N;i++)
  {                                   {U[i],V[i]}
    {U[i],V[i]}        ⟹           for(i=0;i<N;i++)
    {W[i],X[i]}                       {W[i],X[i]}
  }
```

The overall algorithm for loop fission for energy is given in Figure 10.3. The purpose of this algorithm is to try different loop fissioning strategies for a given nested loop. If there are K instructions within the loop nest, the main 'for loop' in the algorithm enumerates $K-1$ alternatives. The ith alternative is formed by breaking up the nest into two nests after the ith statement from the beginning of the nest (if it is legal to do so). We also add two more alternatives to these $K-1$ alternatives: the original nest and a code with K nested loops each with its own statement (again, if this is legal). The $K+1$ alternatives considered by the algorithm in Figure 10.3 are as follows:



For each alternative, the estimated dynamic energy consumption is calculated, and the alternative with the minimum energy consumption is selected. Although we evaluate only a limited set of alternatives, our experimentation indicated that there was no other fissioning strategy for the codes in our experimental suite that would result in more energy savings.

While it is possible to develop a loop fissioning strategy that targets the entire procedure (by taking into account the inter-nest interactions), in this study, we have just focused on the most energy consuming nest (called *dominating nest*) from each benchmark, and applied the loop fissioning algorithm only to that nest. This approach is expected to improve the overall energy consumption, possibly at the expense of an increase in energy consumption of the other nested loops, because in all the codes that we experimented with, there were only one or two dominating nests. If there are two nested loops that consume exactly the same (maximum) amount of energy, we designate the first one (in the textual program order) as the dominating nest.

Figure 10.4 shows the percentage decrease in energy (when the *entire application* is considered) brought about by each fissioning alternative over mode control plus clustering. It should be noted that we consider only the codes that can benefit from loop fission, and different most costly nests (of different codes) have different number of fissioning options (alternatives). The main reason that prevented the application of loop fission to other codes was the fact that the dominating nests in these codes contain only a single instruction; so, there was only one option, which is the original nest. We also observed that applying loop fission to the *second dominating nest* (instead of the dominating nest) in general increased the overall energy consumption as the array layouts suggested by the second most costly nest are usually not suitable for the most costly nest.

Consequently, we selected the most effective alternative for each benchmark, and applied it. The average improvement (%) over all benchmarks

INPUT: *A perfect loop nest containing K instructions*
OUTPUT:
An energy-optimized loop-fissioned code
(B contains the number that identifies the loop fissioned version with minimum energy)
Begin
Let E_{min} *be the least computed energy consumed by the loop nest*
Let B be the number of the more energy saving option.
$E_{min} \leftarrow$ *The energy consumed by the original loop nest (without fission)*
$B \leftarrow 0$
For *i = 1 to K - 1*
Create two loop nests instead of the original loop nest
• *the first containing the first i instructions and the second the remaining* $K - i$ *instructions*
Compute the energy E_i *consumed by this new program.*
If $E_{min} > E_i$
$E_{min} \leftarrow E_i$
$B \leftarrow i$
EndIf
EndFor
Create K loop nests, instead of the original loop nest
• *each one contains one instruction of the original nested loop*
Compute the energy E_K *for this new program*
If $E_{min} > E_K$
$E_{min} \leftarrow E_K$
$B \leftarrow K$
EndIf
End

Figure 10.3. An Algorithm that Improves Memory Energy Consumption using Loop Fission

(that are amenable to loop fission) was around 55.5%. Since the array allocation considers only the dominating nest, there might be a negative impact (energy-wise) on other nests if the best layout strategy for the most costly nest is not suitable for the other nests. Figure 10.5 shows the energy impact of this dominating nest-centric optimization on other nests. We observe that different benchmarks behave quite differently. For example, in `eflux`, all the other nests (in addition to the dominating nest) show energy improvement. For `vpenta`,

Benchmark	Alternative Fissioning Strategies for the Dominating Nest (%)								
	#1	#2	#3	#4	#5	#6	#7	#8	#9
adi	47.0	47.0	61.2						
dtdtz	48.5	48.5	48.5	48.5					
eflux	47.0	45.2	43.3	66.9					
matvec	49.8	33.2	16.6	58.2					
tomcatv	49.5								
vpenta	8.2	23.5	16.6	24.9	18.0	16.6	20.7	8.2	48.4

Figure 10.4. Percentage Energy Improvements due to Different Loop Fission Alternatives

Benchmark	Loop Nests (%)								
	#0	#1	#2	#3	#4	#5	#6	#7	#8
adi	61.2	0.0							
dtdtz	48.5	0.0	-33.0						
eflux	47.0	19.9	24.7	19.9	19.9	25.0			
matvec	58.2	-48.1	0.0	0.0					
tomcatv	49.5	0.0	0.0	0.0	-1.5	0.0	16.3	0.0	0.0
vpenta	-33.2	-39.8	0.0	48.4	-33.3	-33.3	-0.3	25.0	

Figure 10.5. Percentage Variations in Energy of All Nested Loops in the Benchmarks

on the other hand, the second nest experiences a 39.8% increase in energy consumption. Overall, the nests *other than the dominating ones* show only a 2.8% increase in energy. Thus, we conclude that focusing only on the dominating nest and performing an array allocation based on that works well with loop fission.

4.4 Improving Effectiveness of Power Mode Control Mechanisms Using Data Transformations

While loop transformations can be used to improve the effectiveness of low power memory mode operation as shown in previous subsection, they can be problematic to apply in certain cases. For example, unlike other languages such as C or Fortran, changing access pattern of Java codes is more problematic as the language imposes a precise exception requirement. That is, it is not safe to change the execution order of loop iterations unless one is certain that the original occurrence order of exceptions will be preserved after this modification. This is the primary obstacle before any transformational approach based on loop transformations and instigated some efforts in eliminating array bounds checking and in developing constrained loop optimizations [7]. Data transformations on the other hand are independent of access pattern and cannot modify/constrain occurrence order of exceptions. In the following two

subsections, we summarize the anticipated impact of using two data optimization approaches on energy consumption. The details of the techniques can be found in [1] and are omitted here for lack of space.

4.4.1 Data Layout Transformations. The data layout transformations can be used to reduce the energy savings of partitioned memory modules with low power modes. By changing the storage order and improving the spatial locality of the arrays using the transforms, one can expect to increase the average inter-access times between two memory operations to the same module. The increased inter-access times would provide an opportunity for operating in a lower power mode for a longer time. Note that the increased inter-access time could be due to two different reasons. First, improved cache behavior can increase the time between two accesses to memory. Next, the memory accesses can be confined to a particular memory module for a period of time allowing the other modules to operate in a lower power mode for extended periods of time. For example, consider storing a large array that spans across different modules. If it is stored in row-major form and accessed in column-major form, successive references may access different modules. However, if it is stored and accessed in column-major form, successive accesses will be confined to the same module except at module boundaries.

4.4.2 Array Interleaving. In this section, we discuss a data space transformation technique, array interleaving, that transforms a number of array variables simultaneously. It achieves this by interleaving the memory allocation for the elements of a set of arrays in a common heap space. This can be used to eliminate inter-variable conflict misses, that is, the conflict misses that are due to different variables. In our partitioned memory architecture, such an optimization can also maximize opportunities for placing memory modules into a low power operating mode.

For the partitioned-memory architecture, only those memory modules containing the parts of the arrays currently being accessed need to be active. If we use the array interleaving strategy, it would co-locate portions of different arrays which are accessed at the same time. This can provide an opportunity for transitioning more modules into a lower power mode. For instance, let us assume that the values of elements from one array `U` are copied to the elements of another array `V` and that the two arrays are allocated in two different memory modules. When copying the elements of one array to another, we need to keep both modules active for the entire duration of loop execution. After interleaving, the corresponding rows of each array are co-located. Hence, half of each array spans each module. Thus, during the first half of the loop execution one of the modules can be transitioned into a lower power mode and in the second half of the loop execution the other module can operate in a lower power mode.

Through experimentation on a set of Java applications [1], it is observed that using layout transformation provides an average of 9.68% energy saving in addition to the savings provided by mode control [1]. Similarly, the array interleaving optimization provides an average of 14.96% additional energy savings.

5. Summary

In this chapter, we have shown that software optimizations play a critical role in determining the system energy. These optimizations can be applied at different stages of the compiler, either at the high-level or the low-level. Further, we demonstrated that optimizing for energy and performance are not the same. The optimizations proposed also stress the importance of increasing the synergy between the underlying low power hardware and the software executing on it in order to gain significant energy savings. While most of this chapter has focussed on dynamic power consumption, it will be vital in future to focus on leakage power management. Also, it would be important to dynamically generate code that adapts to changing energy constraints in mobile environments.

Acknowledgments

The authors wish to acknowledge the contributions of the students from the Microsystems Design Group who have worked on several projects reported in this paper.

Notes

1. Note that, even if there is no edge between two instructions, we will still have a weight between them, as these two instructions can be executed in the final schedule one after another.

References

[1] R. Athavale, N. Vijaykrishnan, M. Kandemir and M. J. Irwin. Influence of Array Allocation Mechanisms on Memory System Energy. In Proc. of *International Parallel and Distributed Processing Symposium*. April 2001.

[2] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers,* June, 1998.

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

[4] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In

Proc. of *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, November 2000.

[5] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Memory system energy: Influence of hardware-software optimizations. In Proc. of *ACM/IEEE International Symposium on Low Power Electronics and Design*, Rapallo/Portofino Coast, Italy, July, 2000.

[6] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of Compiler Optimizations on System Power. In Proc. of *the 37th Design Automation Conference*, Los Angeles, CA, June 5–9, 2000.

[7] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java Programming for High-Performance Numerical Computing. *IBM Systems Journal*, 39(2), 2000.

[8] M. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *Fujitsu Scientific and Technical Journal*, 31(2), pp. 215–229, 1995.

[9] A. Parikh, M. Kandemir, N. Vijaykrishnan and M. J. Irwin. Instruction scheduling based on energy and performance constraints. In Proc. *The IEEE CS Annual Workshop on VLSI*, Orlando, FL, April 27-28, 2000.

[10] K. Roy and M. C. Johnson. Software design for low power. *Low Power Design in Deep Sub-micron Electronics*, Kluwer Academic Press, October 1996, ed. J. Mermet and W. Nebel, pp. 433–459.

[11] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32 bit embedded processors. In *Proc. ICCD'98*, Austin, TX, October 1998.

[12] C.-L. Su, C.-Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Proc. IEEE COMCON*, 1994, pp. 489–498.

[13] V. Tiwari, S. Malik, A. Wolfe, and T.C. Lee. *Instruction Level Power Analysis and Optimization of Software*, Journal of VLSI Signal Processing Systems, Vol. 13, No. 2, August 1996.

[14] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, December 1994.

[15] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture*, Vancouver, British Columbia, June 2000.

[16] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison Wesley, CA, 1996.

[17] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN 91 Conference Programming Language Design and Implement ation*, pages 30–44, June 1991.

[18] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In Proc. *37th Design Automation Conference*, Los Angeles, June 2000.