Enhancing Compiler Techniques for Memory Energy Optimizations

Joseph Zambreno[1], Mahmut Taylan Kandemir[2], and Alok Choudhary[1]

[1] Department of Electrical and Computer Engineering
Northwestern University
Evanston IL 60208, USA
{zambro1,choudhar}@ece.northwestern.edu
[2] Microsystems Design Lab
Pennsylvania State University
University Park PA 16802, USA
kandemir@cse.psu.edu

Abstract. As both chip densities and clock frequencies steadily rise in modern microprocessors, energy consumption is quickly joining performance as a key design constraint. Power issues are increasingly important in embedded systems, especially those found in portable devices. Much research has focused on the memory subsystems of these devices since they are a leading energy consumer. Compiler optimizations that are traditionally used to increase performance have shown much promise in also reducing cache energy consumption. In this paper we study the interaction between performance-oriented compiler optimizations and memory energy consumption and demonstrate that the best performance optimizations do not necessarily generate the best energy behavior in memory. We also show a simple metric that a power-optimizing compiler can utilize in order to capture the energy impact of potential optimizations. Next, we present heuristic algorithms that determine a suitable optimization strategy given a memory energy upper bound. Finally, we demonstrate that our strategies will gain even more importance in the future when leakage energy is expected to play an even larger role in the total energy consumption equation.

1 Introduction

As the market for embedded systems continues to grow, power consumption issues are becoming increasingly important. In fact, as new cell phones, PDAs, and e-mail devices are being developed, the metric of performance / battery hours is considered crucial [24]. Much research has been done on developing low-power systems and techniques, ranging from circuit-level to architecture to compiler and operating system support. Our research concentrates on the memory subsystem mainly because it is a significant contributor of power consumption in embedded systems [19] and high-performance processors [8].

Current optimizing compilers perform various optimizations for increasing instruction-level parallelism and improving data locality. Many of these compiler optimizations, such as loop unrolling, loop tiling, and function inlining

A. Sangiovanni-Vincentelli and J. Sifakis (Eds.): EMSOFT 2002, LNCS 2491, pp. 364–381, 2002.

tend to increase code size. This increased code size is an important drawback in embedded systems, as many of these systems execute a single or a small set of applications, and application code sizes (executable sizes) are the primary factor that determines instruction memory size. An increase in instruction memory size, in turn, increases both per access dynamic energy consumption and leakage energy. Therefore, in an energy-conscious environment, the aggressiveness of compiler optimizations must be tuned carefully to keep instruction memory energy consumption under control.

In this paper, we investigate the effect of performance-oriented compiler optimizations on the memory energy consumption. We first analyze the tradeoffs between code size and performance by compiling several benchmark programs with different performance-oriented optimizations. Using analytical SRAM energy dissipation models, we investigate how an increased code size increases the memory energy consumption due to instruction accesses. In doing so, we also illustrate that a simple metric can be used as a first-degree estimate of instruction and data memory energy. Our second contribution is a compiler algorithm that determines a suitable optimization strategy for a given memory energy constraint. We study the effectiveness of our strategy in reducing memory energy for both loop unrolling and function inlining, and examine a futuristic scenario where leakage energy constitutes a sizeable portion of the overall memory energy budget. Note that the leakage energy consumption is particularly important in large SRAM memories that are active throughout execution and all trends [5] indicate that it will be much more important in upcoming process technologies.

Our experimental results emphasize the importance of taking into account the energy impact of optimizations early in the design process, and show that an energy-conscious function inlining algorithm can reduce energy consumed in memory by as much as 30% as compared to an aggressive performance-oriented inlining strategy, with comparable results for our loop unrolling strategy. Based on our results, we conclude that loop unrolling and function inlining are two optimizations that illustrate the tradeoff between performance and energy.

The remainder of this paper is organized as follows. Section 2 discusses related work in low-power research and the contribution of this paper within that framework. Section 3 presents results detailing the effects of some standard compiler optimizations on performance and resulting code size. Section 4 presents and analyzes energy-conscious heuristics for loop unrolling and function inlining. Section 5 reports on the energy impact of our approach when leakage energy is taken into account. Finally, Sect. 6 concludes the paper by summarizing our contributions and giving an outline of the planned future research on this topic.

2 Related Work

We discuss the related research in the field of low-power computing as it fits into three categories. At the *circuit-level*, there have been numerous optimizations proposed for minimizing energy consumption. Powell et al. [22] present a gated supply voltage design that interacts with a dynamically resizable instruction

cache. By turning off the supply voltage to unused sections of the cache, their method effectively eliminates the leakage power consumption in those sections. Ye et al. [30] developed a method of transistor stacking in order to reduce leakage energy consumption while maintaining high performance. Chandrakasan and Brodersen [5] present several techniques on low-power circuit design.

At the *architectural-level*, much work that has been done to improve memory and CPU performance with the added expectation that power consumption will also improve. Also in this area several techniques have been proposed to reduce switching and leakage energy consumption at the cost of small performance losses. Hajj et al. [9] present instruction cache energy reduction by using an intermediate cache between the instruction cache and main memory. Their research shows that this smaller intermediate cache allows the main instruction cache to remain disabled most of the time. Delaluz, Kandemir, et al. [7] discuss using low-power operating modes for DRAMs to conserve energy consumption by effectively shutting off the DRAM when not in use. They present compilation techniques to analyze and exploit memory idleness and also a method by which the memory system can use self-detection to switch to a lower-power operating mode. In [1], Balasubramonian et al. suggest a cache and TLB layout that significantly decreases energy consumption while increasing performance. Their suggested layout allows for a dynamic memory configuration that analyzes size and speed tradeoffs on a per-application basis. Kaxiras, Hu, and Martonosi [13] present a method to reduce cache leakage energy consumption by turning off cache lines that likely will not be used again. By realizing that most cache lines typically have a flurry of frequent use when first introduced and then a period of "dead time" before they are evicted, Kaxiras et al. were able to reduce L1 cache leakage energy by 5× for certain benchmarks with only a negligible performance decrease.

At the *software-level*, many preliminary investigations have been conducted into compiler techniques, more specifically to analyze how optimizations developed to increase performance can also improve energy consumption. In [4], Catthoor et al. offer a methodology for analyzing the effect of compiler optimizations on memory power consumption. Mehta et al. [17] investigate the effect of loop unrolling, software pipelining and recursion elimination on CPU energy consumption. They also present an algorithm for register relabeling that attempts to minimize the energy consumption of the register file decoder and instruction register by reducing the amount of switching in those structures. An introductory look into other high-level optimizations such as loop fusion, loop distribution and scalar replacement is performed with SimplePower in [26]. Hajj et al. examine function inlining in [9], but only in the context of its effectiveness with custom cache architectural modifications. In [14], Ellis et al. propose an integrated hardware/software approach for exploiting a power-aware memory hierarchy. Ramanujam et al. [23] present an algorithm to estimate the actual memory requirement for data transfers in embedded systems. They also present loop transformations that attempt to minimize the amount of memory required. In [10], Halambi et al. investigate a novel compiler technique to reduce

the bit-width of instructions to reduce code size. Muchnick [21], Morgan [20], and Leupers [16] propose techniques for limiting the aggressiveness of function inlining. Our work is different from theirs in a number of ways: first, we focus on energy consumption; second, we present a metric that captures the energy behavior of the applications being optimized; and third, in addition to inlining, we also study other classical performance-oriented techniques.

Before developing a low-power technique, a predetermined method of estimating its effectiveness is required. Most research in this field leverages cycle-level simulators. Much work has been done on extending the popular SimpleScalar simulator [3] to include power-estimation models. As an example, both the Wattch simulator [2] and the SimplePower simulator [26] leverage the SimpleScalar framework to model power consumption in a standard 5-stage pipelined RISC datapath. The SimplePower simulator uses a table-lookup system based on power models for memory and functional units, while Wattch relies on more detailed parameterized models. Although these simulators provide detailed analysis of the energy consumption in the major system components, they are not primarily meant for investigating compiler optimizations. Also in this category are tools and methods that give run-time energy estimates. For example, the Castle tool [11] profiles hardware performance counters and feeds that data into energy models to estimate the overall consumption in the main CPU components. Kamble and Ghose [12] derive analytical cache energy dissipation models and verify them against a low-level simulator. The models in [12] are used to investigate architectural-level cache changes. In contrast, in this work, we exclusively study the impact of code optimizations on energy and performance.

3 Analyzing Performance and Energy Tradeoffs

We focus on a System-on-Chip (SoC) design where we have both an instruction memory and data memory. We also assume the existence of a data cache and a larger off-chip data memory. Consequently, data locality optimizations [29] are vital to take advantage of the small on-chip data memory structures. As with other architectures, it is also important to increase instruction level parallelism (ILP) as much as possible. This is particularly important in environments that process digital signal processing applications as many DSP codes have high ILP requirements.

The dynamic energy consumption in instruction memory during the execution of an application depends on two factors: the size of the instruction memory and the number of accesses to the instruction memory [4]. Applying aggressive performance oriented optimizations can increase both these factors. The size of the instruction memory can increase due to the fact that many compiler optimizations such as function inlining, procedure cloning, iteration space tiling, and loop unrolling increase code size. The number of instruction memory accesses can increase due to the fact that instruction reuse is decreased after most performance-oriented compiler optimizations for data locality [21]. In this section, we investigate the effect of various standard compiler optimizations on performance, executable size, and energy consumption.

3.1 Estimating Energy Consumption

In order to obtain the energy consumption results when compiler optimizations are applied, we have enhanced the analytical models for cache energy dissipation found in [12] to model energy consumption in instruction and data memories. These models use the run-time data from the cache such as hit/miss counts, data and address bit widths, and switching probabilities in order to estimate the energy consumption in its major components. The overall cache energy consumption is determined by this run-time data and also by the specifics of its organization, such as cache size, block size, and associativity.

We have changed these models to reflect an on-chip memory hierarchy such as would be found in an embedded SoC architecture, with an instruction memory, a data cache, and a data memory. In many embedded devices, the memory is of a preset size that is determined by the fixed applications that run on it. That is, the memory size is chosen by taking the executable size into account, plus a small amount of space for temporary variables. For our experiments, we set the memory size equal to the code size of a given benchmark. There are numerous capacitive coefficients that need to be evaluated in order to use our model. These values come from the data for the 0.8μ transistor implementation found in [27]. A memory power supply of 3.3 V is assumed, although for relative energy calculations its value is unimportant.

3.2 Methodology

We measured the effect of compiler optimizations using benchmarks from the SPEC CPU2000 [25] and MediaBench [15] suites. The chosen benchmarks from the MediaBench suite perform audio/video encoding and decoding and are similar to the tasks performed by typical embedded processors in multimedia devices. The SPEC benchmarks, while not normally considered to be indicative of an embedded workload, nonetheless have interesting locality characteristics and make for a good comparison to their MediaBench brethren.

To perform our experiments we decided to leverage a pre-existing optimizing compiler, the MIPSPro compiler from Silicon Graphics, Inc. The MIPSPro compiler allows us to pick and apply both loop nest optimizations and interprocedural optimizations by using compiler directives and/or setting runtime parameters. There are four major modes [18] of the MIPSPro compiler that perform different performance optimizations:

–**O0:** No code optimization is done.
–**O1:** Performs copy propagation, dead code elimination, and other local optimizations.
–**O2:** Performs non-loop if conversion, with some cross-iteration optimizations (no write/write elimination on loops without trip counts). This mode also performs loop unrolling and recurrence fixing. Basic blocks are reordered to minimize the number of taken branches.
–**O3:** Performs more if conversion and software pipelining. This mode also activates the Loop Nest Optimizer (LNO) that attempts locality-enhancing optimizations such as tiling, fission/fusion, and loop interchange [21,29].

Used in conjunction with these four optimization modes is the –**IPA** flag that turns on the interprocedural analysis optimizations, which include function inlining, interprocedural constant propagation, and dead function elimination. More details on these optimizations can be found in [18,28,29,21].

We also needed an accurate way to estimate the values of the run-time data for our target embedded architecture. The MIPS R10000 we were compiling on contains several relevant hardware counters that we were able to sample by using the SGI performance counter profiler tool, *perfex* [18]. Of course the R10000 is a general-purpose processor, with several advanced features that would most likely not be present in an embedded CPU core (L2 cache, out-of-order execution, multi-instruction issuing, etc.). We were able to overcome this obstacle by carefully choosing which hardware counters to profile, ignoring some statistics all together (L2 cache hit/miss rates), and using other data (percentage of speculated instructions) to mask the modern features of the R10000. In the end, we plugged the run-time data and memory size data into our analytical energy equations and estimated the energy output for a given code optimization.

3.3 Code Size/Performance Analysis

Figure 1 shows the resultant code size and dynamic instruction count for three of our benchmarks compiled using the various MIPSPro options. These results are normalized with respect to –O0. From these results, we can observe several trends. First, with the interprocedural analyzer turned off, each optimization mode from –O1 to –O2 shows (in general) both a smaller code size and a smaller dynamic instruction count. This is due to the fact that these levels perform many optimizations that either remove unnecessary code or optimize for performance without adding code. The largest benchmark, *mesa*, shows the least change in code size for all of the optimization modes. Even though it executes billions of instructions, the *equake* benchmark has a relatively small unoptimized code size, and the optimizations are very successful at decreasing the code size. For these benchmarks, there is a trend that shows that the larger the original (unoptimized) code size, the less effect these optimizations have on decreasing that code size. The –O2 optimization level leads to the smallest code size, on average a 10% improvement over no optimizations at all.

Second, at the –O3 optimization level, the loop nest optimizer performs more aggressive loop unrolling along with other trade-offing optimizations, and the results are mixed. For the benchmark codes in our experimental suite, running the LNO at the –O3 level leads to on average a 1% performance increase over the –O2 level across all benchmarks used, at the cost of a 11% increase in code size. This small performance improvement is due to the fact that some of our benchmarks do not contain too many regular nested loop structures to take full advantage of the aggressive optimizations in the LNO option. The –O3 optimization level leads to the best overall performance, the average benchmark running in 52% of the time of its unoptimized counterpart. On average, the optimizations have more of an effect on instruction count than they do on code size. However, there

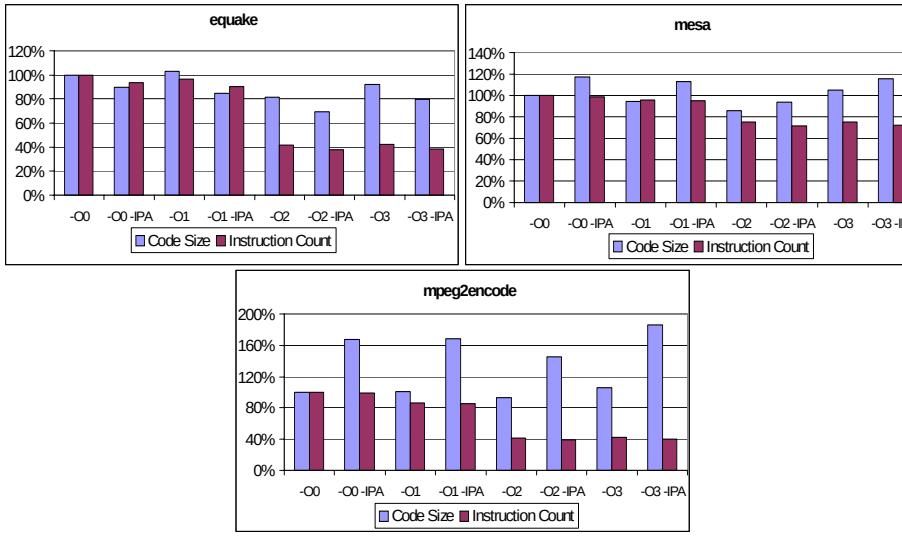

Fig. 1. Normalized code size and instruction count for MIPSPro optimization settings. These results show that the –O2 optimization level leads to the smallest code size on average, while the –O3 –IPA optimization level leads to the best performance on average. These results clearly show the tradeoff between code size and performance

does not appear to be a trend between the number of unoptimized instructions executed and the effect of the optimizations on that instruction count.

Third, we see that adding the –IPA option leads to more interesting results. Many of the optimizations in this group, most notably function inlining, generally increase the code (executable) size. On average, compiling with the –IPA flag leads to a code size that is 19% larger as compared to the same level of optimization without interprocedural analysis. With this penalty comes on average a 7% improvement in performance. The MediaBench benchmarks, which have a lower unoptimized instruction count than their SPEC counterparts, show a much smaller effect of the interprocedural analysis on performance. For example, compiling with the –IPA flag on the *mpeg2encode* benchmark leads to on average a 1% performance increase. We also observe that the most sophisticated optimization level (that is, –O3 –IPA) generates an average performance improvement of 55% and increases the executable size by 25% as compared to the original unoptimized code. Since the instruction memory energy consumption is proportional to the executable size, these results clearly show the tradeoff between memory energy and performance.

Therefore, an important question now is to determine an optimization strategy that gives an acceptable performance without too much of an increase in energy consumption. Later in Sect. 4, we present a heuristic approach for addressing this problem. In the following, we present a simple metric that allows an optimizing compiler to estimate instruction memory energy consumption without actually using an energy estimation tool.

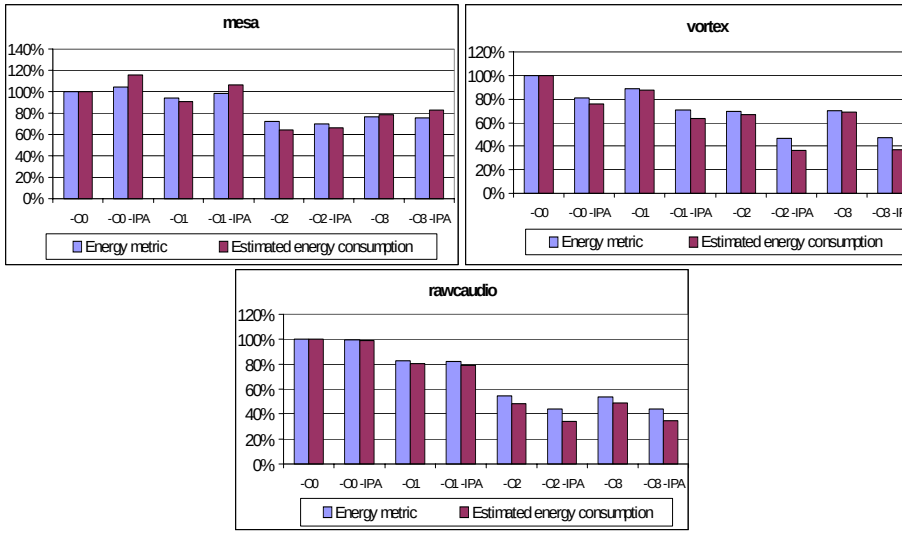

Fig. 2. Normalized energy metric values compared to normalized calculated energy consumption. These results show that the –O2 –IPA optimization level leads to the lowest energy on average, and that the energy metric shows similar trends to the analytically determined values (within 9% on average)

3.4 Analyzing Energy Consumption

In general, the per-access energy cost is directly related to the memory size, which in embedded devices is determined by the number of bytes required to store its code or data. Also, the total number of instructions executed is an accurate measure of the number of times that an instruction memory would need to be accessed. For this reason, we explored using the product of the code size and the instruction count as an early estimate to how much energy a given benchmark would be consuming in instruction memory. Similarly, for data memory, the energy consumption would be dependent on the number of accesses and the data size. In practice profiling can be utilized to find values for the instructions executed and number of data accesses. We used profiling to find the instruction count, but for simplification purposes we estimated the data access count by assuming a constant ratio (30% is a common choice) of data accesses to total instructions.

Figure 2 depicts the effect of the MIPSPro optimizations on the sum of these two metrics and compares the observed trends with those obtained through actual energy calculations. We see from these graphs that both our metric and actual energy calculations indicate that the –O2 –IPA option is the most energy-efficient one. In other words, using the most aggressive optimization strategy (–O3 –IPA) may not be the best choice from the energy perspective. We also observe that turning on the –IPA option appears to help the –O2 optimization mode more than others.

The results shown in Fig. 2 clearly indicate that the energy trends estimated when our metric is employed are similar to those obtained when actual energy calculations are carried out. For example, both approaches indicate that the –O2 optimization level with the interprocedural analysis turned on leads to the best energy conservation, consuming on average only 46% of the energy of the unoptimized benchmarks. More importantly, the relative estimated energy consumption shows very similar trends to our metric of $IC \cdot (CS + 0.3 \cdot DS)$. The metric is a good predictor of the relative energy consumed, on average being within 9% of the energy estimate.

Based on these observations, we conclude that a compiler optimization technique that minimizes the metric of $IC \cdot (CS + 0.3 \cdot DS)$ will also minimize the memory energy consumption in most cases. That is, the $IC \cdot (CS + 0.3 \cdot DS)$ metric can be utilized to rank the memory energy consumptions of different optimized versions of a given embedded code. This is an important conclusion as it indicates that, instead of using complex energy calculations, a compiler can adopt an energy estimation strategy based on the estimation of dynamic instruction count along with executable and data size. Also, previous work [28] shows that accurately estimating static and dynamic instruction count at compile time is possible even for sophisticated superscalar processors such as the MIPS R10000. Therefore, such estimates can be used for obtaining an idea about instruction and data memory energy consumption of a given code under a set of optimizations.

However, in many cases, instead of trying to reduce energy consumption as much as possible (at the expense of performance), it might be more important to compile a given application under a memory energy constraint. The following section addresses this issue, and proposes a heuristic technique that can easily be employed by an embedded compiler that targets both energy and performance.

4 Energy-Constrained Compiling

Since overly aggressive loop restructuring and interprocedural optimizations can lead to an undesirable tradeoff between energy consumption and performance, it is of interest to investigate tailoring these optimizations to fit to energy constraints. In this section we present and analyze heuristics that provide a systematic way of choosing a suitable loop unrolling strategy that attempts to improve performance while keeping in mind energy consumption. We then provide the same treatment to a heuristic for function inlining.

4.1 Energy-Aware Loop Unrolling Heuristic

Loop unrolling is a commonly used optimization whereby the loop body is replaced by several copies of the loop body [21]. The main performance benefit of loop unrolling is the removal of the execution of many of the branches found in the loop iteration limit test code. Unrolling code also has the potential to improve the effectiveness of other optimizations such as software pipelining. For

```
ENERGY_UNROLL(C, E_limit) {
    C_new = C;
    unroll_factor = 1;
    repeat {
        C_old = C_new;
        l_unroll = ∅;
        L = loop_list(C_old);
        for each loop l ∈ L do {
            if (loop_size(l) < unroll_factor) then {
                l_unroll = l;
                break;
            }
        }
        if (l_unroll ≠ ∅) then {
            C_new = perform_unrolling(C_old, l_unroll, unroll_factor);
            E = estimate_energy(C_new);
        }
        else {
            unroll_factor = unroll_factor * 2;
        }
    }
    until (E > E_limit);
    return(C_old);
}
```

Fig. 3. Energy-aware loop unrolling heuristic

these reasons it is generally expected that applying loop unrolling will lead to both performance and energy improvements. However, the unrolled version of a code is in general larger than the original version. Consequently, loop unrolling may increase the per-access energy cost of instruction memory, and it would be important for an embedded compiler to be careful in applying unrolling to leverage the performance gains while limiting the increase in energy consumption.

Figure 3 shows our energy-aware unrolling heuristic. Our approach to the problem is as follows. We start with an unoptimized program and a set of loops inside the function that we are interested in unrolling. The value of the unrolling factor n is set to 1 in the initial iteration. At each step, we choose the first loop that has not been unrolled yet by a factor of n, and then unroll it by that amount. After the unrolling, we estimate the resultant energy consumption and compare it with the upper bound. If the upper bound has not been reached, we select the next loop to unroll. If all the desirable loops have already been unrolled by a factor of n or more, we increment n and repeat the process. Once the energy consumption after an unrolling becomes larger than the upper bound, we undo the last optimization and return the resulting code as the output.

Figure 4 shows the performance (in terms of graduated instruction count) for the *mesa* benchmark when our unrolling heuristic is applied under different

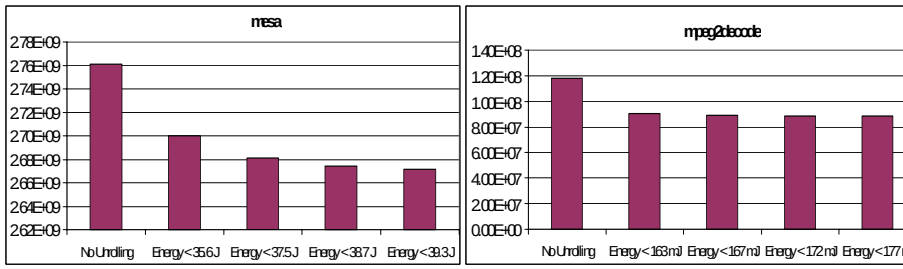

Fig. 4. Instruction count for the *mesa* and *mpeg2decode* benchmarks as a function of the energy upper-bound of our loop unrolling heuristic algorithm. These results show that by carefully choosing our energy upper bound, we are able to leverage the performance gains of unrolling while saving energy consumption when compared to the more aggressive unrolling strategies

memory energy upper bounds. The benchmark is compiled at the –O2 MIPSPro optimization mode with unrolling turned off as the base case. It can be observed that we achieve a performance improvement of 14% on the average across all energy bounds used. As seen in the second bar in each subgraph in Fig. 4, in the most restrictive case our heuristic algorithm is able on average to provide a 10% energy consumption savings while keeping performance to within 2% of the more aggressive unrolling strategies.

4.2 Energy-Aware Function Inlining Heuristic

Function inlining is an interprocedural compiler optimization whereby a copy of the code for a procedure call is replicated at a call site [29]. There are two major reasons that an optimizing compiler might apply inlining. First, applying inlining eliminates the overhead associated with a procedure call (saving and restoring register values, passing parameters, etc.). Second, inlining exposes the function code to the calling environment enabling subsequent code/data optimizations. For this reason, function inlining tends to increase performance and is employed by many commercial compilers. The main disadvantage to function inlining is that it generally increases code size, which, as we showed in Sect. 3, can lead to an undesirable instruction memory energy increase in embedded systems. Also, aggressive inlining can increase the number of local variables, potentially creating more register spills and increasing data memory energy. Therefore, an optimizing compiler designed for embedded environments should be careful in applying inlining and should try to strike a balance between increased performance and increased energy consumption. In particular, it is important to maximize the performance (through inlining) while keeping the increase in instruction memory energy under control.

Figure 5 shows our energy-aware inlining heuristic. Our approach to this problem is as follows. We start with the unoptimized program and, at each step, try to select the most appropriate (function, call-site) pair and perform

```
ENERGY_INLINE(C, E_limit) {
    C_new = C;
    repeat {
        C_old = C_new;
        F = function_list(C_old);
        if |F| < 2 then {
            return (C_old);
        }
        else {
            [f_inline, cs_inline] = best inlinable candidate ∈ F;
            C_new = perform_inlining(C_old, f_inline, cs_inline);
        }
    }
    until (E > E_limit);
    return(C_old);
}
```

Fig. 5. Energy-aware function inlining heuristic

inlining. After an inlining is performed, we estimate or measure the resulting energy consumption and compare it with the energy upper bound. If we are still under the upper bound, we select the next (function, call-site) pair, and so on. The process stops when the energy consumption after the inlining becomes larger than the upper bound. When this occurs, we undo the last inlining, and return the resulting code as the output. This algorithm does not attempt to find an optimal inlining strategy in terms of energy consumption; it just guarantees that a specified upper-limit is not exceeded.

Note that our heuristic is not specific about how the most appropriate (function, call-site) pairs are chosen and in practice, there are several methods that can be used in order to select these pairs. The simplest approach is by brute force. An optimizing compiler can iterate through every possible function call to choose the one that gives the most performance benefit when inlined. This approach, while not very efficient, works well for applications with relatively small numbers of (function, call-site) pairs. A more advanced approach involves using an execution-profiling tool such as the SpeedShop tool on SGI machines [18]. By measuring hardware counters such as the number of graduated instructions, a profiling tool can provide hints to an optimizing compiler about call-sites where a large percentage of total instructions are executed. Many of these tools can also generate basic-block profiling runs, where the exact number of times that a function is called from a particular call-site can be exposed to the compiler. In practice, any of these methods can be used separately or in some combination to find appropriate (function, call-site) pairs for our inlining heuristic algorithm. For our experiments, we used the SpeedShop tool to generate lists of functions where inlining was potentially beneficial, and then we applied the brute force approach to select the most appropriate (function, call-site) pairs.

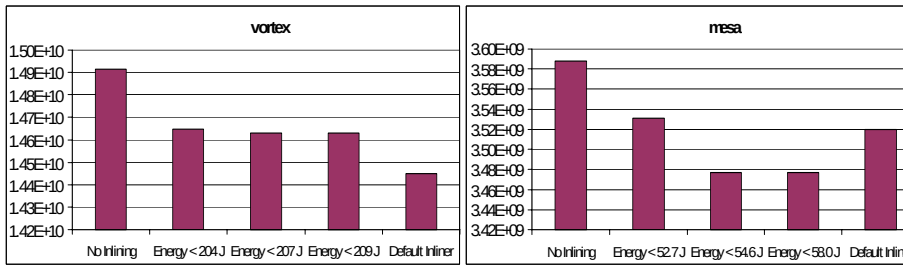

Fig. 6. Instruction count for the *vortex* and *mesa* benchmarks as a function of the energy upper bound of our function inlining heuristic algorithm. These results show that as we relax the energy upper bound, we see a diminishing benefit in terms of performance improvements. Also, these results show that our heuristic will produce very competitive code when compared to the MIPSPro default inliner

Figure 6 shows the energy consumption and performance (execution time) in terms of graduated instruction count for the *vortex* and *mesa* benchmarks compiled with the –O2 MIPSPro optimization mode when our inlining heuristic is applied under different memory energy upper bounds. It can be observed that we achieve a performance improvement of 10% on the average across all energy bounds used. It can also be seen that as we increase our energy upper bound (i.e., allow more energy consumption in memory), our function inlining strategy produces better-performing code, demonstrating the tradeoff between energy and performance. Comparing our heurstic to the default inliner of the MIPSPro compiler, it can be noted that our heuristic reduces the overall memory energy consumption while producing code that is within a few percent of the performance. For the *mesa* benchmark, the default inliner leads to a code that consumes over 83 J while our heuristic chooses an inlining strategy that consumes just under 58 J and performs slightly better. For the *vortex* benchmark, our heuristic consumes 70 J less than the default inliner, with a performance that manages to stay within 2%. The results given in Fig. 4 and 6 indicate that our heuristics improve performance while keeping the energy consumption in memory below a pre-set limit.

5 Leakage Energy

Energy consumption has two major components: dynamic energy and static (leakage) energy. While in CMOS circuits dynamic energy is the dominant part, the current trends indicate that the contribution of leakage energy to the overall energy budget will increase exponentially in upcoming circuit generations [5]. For example, recent energy estimates for 0.13μ process indicate that leakage energy accounts for 30% of L1 cache and as much as 80% of L2 cache energy [22]. Note that leakage energy is consumed as long as the circuit is powered on independent of whether it is actually accessed or not. This is in contrast to dynamic energy,

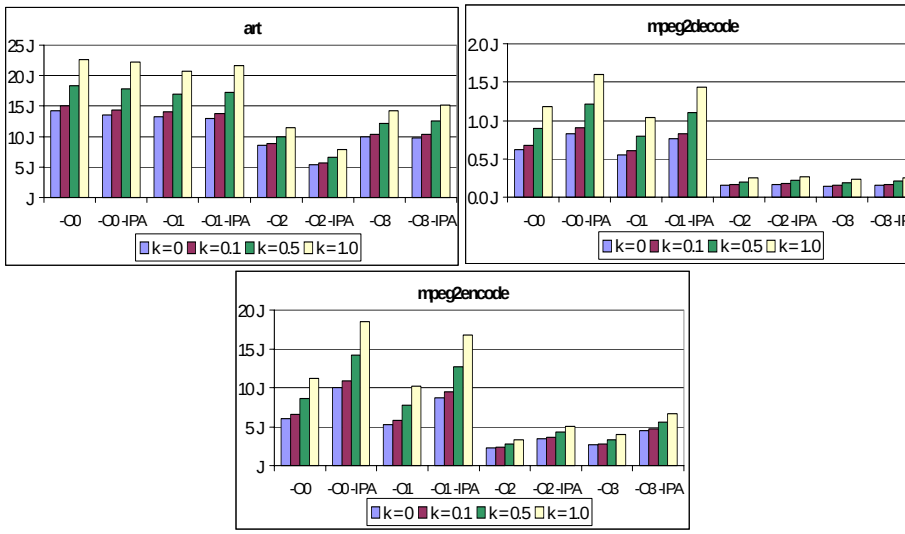

Fig. 7. Total energy consumption (as the sum of both dynamic and leakage energy) for the MIPSPro optimization modes. The value of k refers to the relative weight of the per-cycle leakage energy to the per-access dynamic energy. These results show that as leakage energy begins to dominate the total energy consumption equation, optimizations that lead to code growth lead to an even greater overall energy increase

which is spent only when there is an access. The leakage energy consumption of large SRAM memories is expected to be particularly problematic due to the fact that it increases with the size of memory.

In this section, we first show the impact of taking leakage into account on the tradeoff between memory energy and performance of loop restructuring and interprocedural optimizations. After that, we show how our energy-sensitive compilation approach in Sect. 4 performs when leakage is accounted for. Due to the fact that absolute values for leakage and dynamic energy consumption are closely tied to fabrication processes, in this work, we concentrate on the relative weight of leakage energy to dynamic energy. A common approximation is to take the per-cycle leakage energy consumption to be a ratio of the per-access dynamic energy consumption. We represent that ratio as a nonnegative value k where a smaller k value ($0.1 \leq k \leq 0.2$) represents current fabrication technologies and larger k values ($0.5 < k \leq 1.0$) represent a futuristic scenario where the effect of leakage energy will begin to outpace that of switching energy. Note that similar approaches have been used by previous research as well (e.g., [6]).

Figure 7 shows our results when we compile three of our benchmarks using the major optimization modes of the MIPSPro compiler when considering leakage energy. These results show an increasingly pronounced tradeoff between performance and energy. For example, compiling with the –O0 –IPA optimization mode leads to a performance/energy tradeoff when compared to un-optimized

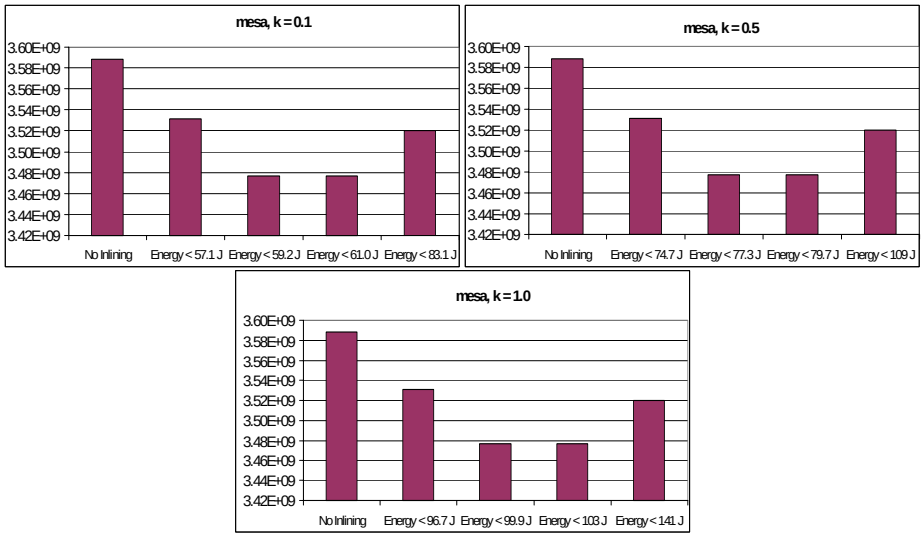

Fig. 8. Graduated instruction count versus memory energy upper bound for different values of k for the *mesa* benchmark. These results show that as we increase the relative weight of leakage energy, we require a greater upper energy bound to achieve the same performance via function inlining

code. For the *mpeg2encode* benchmark, when $k = 0.1$, there is a 1.5% performance improvement that is offset by a 4.3 J energy consumption increase. When $k = 1.0$, there is an equal performance boost that is now offset by a 7.3 J energy consumption increase. This can be explained by the fact that as we increase the relative weight of leakage energy, optimizations that lead to code growth lead to an even greater overall energy increase.

Figure 8 shows our results when we apply our function inlining algorithm to the *mesa* benchmark with an energy upper bound that takes leakage energy into account. For this experiment, we plotted the minimum energy upper bound that would be required for certain performance improvements. The results in Fig. 8 show that as we increase k, we require a greater upper energy bound to achieve the same performance via function inlining. For example, to achieve a performance improvement of 3.1% (as represented by the middle bar of each subgraph of Fig. 8), there is an almost 40 J difference between energy upper bounds when $k = 0.1$ and $k = 1.0$. Clearly, as leakage energy becomes more of a factor in future design fabrication methodologies, it will become increasingly important to limit code and energy growth at the expense of performance gains.

6 Conclusions and Future Work

Power consumption is an important design focus for embedded and portable systems. In this work, we have provided a systematic methodology for analyzing

the effect of compiler optimizations on memory energy consumption. Realizing that many performance-oriented optimizations lead to a tradeoff between increased performance and larger code sizes, we have shown that the product of instruction count and code size is a fairly accurate energy measurement. We have also presented heuristics that attempt to tailor the aggressiveness of both loop unrolling and function inlining, and have analyzed their effectiveness. Finally, we have clearly demonstrated that our techniques will gain greater importance in future design generations, where leakage energy will constitute a larger percentage of the overall memory energy budget, and consequently optimizations that increase code size will increase instruction memory energy consumption by a greater factor.

The results in this work can be exploited in several ways. First, since embedded systems can tolerate much larger compilation times than their general purpose counterparts (as many of them run a single application for which a large number of processor cycles can be spent in compilation), we can run different optimized versions of the code and select the one with the best energy efficiency. Our results presented in this paper indicate that the compiler should attempt to minimize the product of the code size and dynamic instruction count since it is a very good first level approximation for instruction memory energy consumption. Second, systems with strict energy requirements can utilize our function inlining and loop unrolling heuristics to improve performance without violating design constraints. We have shown that the interprocedural optimizations such as function inlining can often lead to dramatic energy consumption increases, while the loop transforming optimizations, if applied intelligently, can lead to energy decreases alongside performance gains. Since interprocedural optimizations are often applied before their loop transforming counterparts, in order to maximize performance it would make sense to allow for the interprocedural optimizations to increase energy a certain percentage past the desired maximum, as the application of the loop transformations will be able to bring the energy consumption level back down to the limit.

In this paper, we introduced concepts and techniques that deal with embedded processor and more specifically Systems-on-Chip (SoCs). In the future we plan on extending our analysis to more generic SoCs, where perhaps embedded CPU cores, FPGAs, and ASICs are incorporated along with numerous memory devices. These heterogeneous SoCs require complicated interconnection protocols, an example being the AMBA bus standard. Optimizations that are meant to improve cache performance gain greater importance in an AMBA configuration, as a cache miss from the CPU can lead to lengthy latencies due to contention on the bus from other on-chip resources. Consequently our future work will include adapting our energy model to reflect the AMBA bus and analyzing how our loop restructuring and interprocedural optimizations effect the energy consumption of different SoC configurations.

Acknowledgement

This work was supported by a National Science Foundation Graduate Research Fellowship.

References

1. R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose architectures. In *Proc. of the 33rd Int'l Symposium on Microarchitecture (MICRO)*, 2000.
2. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Int'l Symposium on Computer Architecture (ISCA)*, 2000.
3. D. Burger and T. Austin The SimpleScalar tool set, version 2.0. Techincal Report CS-TR-97-1342. University of Wisconsin-Madison, 1997.
4. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer Academic Publishers, 1995.
5. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*, Kluwer Academic Publishers, 1995.
6. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning Garbage Collection in an embedded Java environment. In *Proc. of the 8th Int'l Symposium on High Performance Computer Architecture (HPCA)*, 2002.
7. V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proc. of the 7th Int'l Symposium on High Performance Computer Architecture (HPCA)*, 2001.
8. J. Edmondon et al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 1995.
9. N. B. I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proc. of Int'l Symposium on Low-Power Electronics and Design (ISLPED)*, 1998.
10. A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. An efficient compiler technique for code size reduction using reduced bit-width ISAs. In *Proc. of Design, Automation, and Test in Europe (DATE)*, 2001.
11. R. Joseph and M. Martonosi. Run-time power estimation in high-performance micrprocessors. In *Proc. of Int'l Symposium on Low-Power Electronics and Design (ISLPED)*, 2001.
12. M. B. Kamble and K. Ghose. Analytical energy dissipation models for low-power caches. In *Proc. of Int'l Symposium on Low-Power Electronics and Design (ISLPED)*, 1995.
13. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. of the 28th Int'l Symposium on Computer Architecture (ISCA)*, 2001.
14. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power-aware page allocation. In *Proc. of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

15. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *Proc. of the 30th Int'l Symposium on Microarchitecture (MICRO)*, 1997.
16. R. Leupers. *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000.
17. H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low-energy software. In *Proc. of the 25th Int'l Symposium on Computer Architecture (ISCA)*, 1998.
18. *MIPSpro compiling and performance tuning guide.* Silicon Graphics, Inc., 1999.
19. J. Montanaro et al. A 160-MHz, 32-b, 0.5W CMOS RISC microprocessor. *Digital Technical Journal*, 1996.
20. R. Morgan. *Building an Optimizing Compiler*, Butterworth-Heinemann, 1998.
21. S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.
22. M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. of Int'l Symposium on Low-Power Electronics and Design (ISLPED)*, 2001.
23. J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In *Proc. of the 38th Design Automation Conference (DAC)*, 2001.
24. D. Singh and V. Tiwari. Power challenges in the Internet world. In *Cool Chips Tutorial: An Industrial Perspective on Low-Power Processor Desgin* (held in conjunction with *The 32nd Int'l Symposium on Microarchitecture (MICRO)*, 1999.
25. The Standard Performance Evaluation Corporation. http://www.spec.org.
26. N. Vijaykrishnan, M. Kandemir, M. H. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. of the 27th Int'l Symposium on Computer Architecture (ISCA)*, 2000.
27. S. E. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. In *DEC WRL Research Report 93/5*, 1994.
28. M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the 30th Int'l Symposium on Microarchitecture (MICRO)*, 1997.
29. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.
30. Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high-performance circuits. In *IEEE Symposium on VLSI Circuits*, 1998.