35
Low Power/Energy Compiler Optimizations

Ulrich Kremer
Department of Computer Science
Rutgers University

35.1 Introduction

Embedded processor and SoCs are used in many devices, ranging from pace makers, sensors, phones, and PDAs, to general-purpose handheld computers and laptops. Each of these devices has their own requirements for performance, power dissipation, and energy usage, and typically implements a particular tradeoff among these entities. Allowing components of these devices to be controlled by software has opened up opportunities for compilation and operating strategies to reduce power dissipation and energy usage, at the potential cost of performance degradation. Such control includes (1) hibernation, i.e., initiating transitions of a component between a high power, active states and lower power, hibernating states, (2) dynamic frequency and voltage scaling, which allows the clock speed and supply voltage to be set explicitly within a range of feasible voltage and frequency combinations, and (3) remote task mapping, where power and energy is saved on a mobile device by executing a task remotely on a server. In this chapter, we discuss general issues and challenges related to compilers for power and energy management. A set of compilation strategies will be further examined together with initial results that show their potential benefits.

35.2 Why Compilers?

Compilers translate a program in a high-level language into a program that can be executed on a target architecture. In other words, compilers support high-level programming models that allow programmers to describe the solution to their problem at an abstraction level closer to the particular problem domain. As a result, programs are easier to understand and maintain. Porting a program to another target system requires recompilation on the new system rather than reimplementing the program in the new assembly/machine language. However, these benefits may come at the price of a reduction in overall program performance. Typically, the effectiveness of a compiler and its generated code is measured by comparing it against a code that an "expert" assembly/machine code programmer would have written, or even the best machine code possible. For an optimizing compiler, this difference should not be too large, where the acceptable performance gap depends on the particular application domain. What such a comparison does not capture is the effort needed by an "expert" programmer to come up with such a high quality code. Modern embedded processors have many features previously found only in high performance processors, including SIMD instructions, VLIW design, and multiple independent memory banks. The effort to write efficient or even correct programs may be prohibitively high, in particular for embedded systems with short time-to-market cycles. As a result, high-level languages and their optimizing compilers are becoming a necessary alternative to programming advanced embedded processors in machine and assembly code. Instead of rewriting a set of applications for a new target system, a new compiler has to be provided for that new architecture. Researchers in the embedded systems compiler community have developed and are further investigating new compilation infrastructures that allow the effective retargeting of compilers [9]. Although the issue of retargetability is very important, it will not be covered in this chapter.

Optimizing compilers perform program analyses and transformations at different levels of program abstraction, ranging from source code, intermediate code such as three address code, to assembly and machine code. Analyses and transformations can have different scopes. They can be performed within a single basic block (local), across basic blocks but within a procedure (global), or across procedure boundaries (interprocedural). Traditionally, optimizing compilers try to reduce overall program execution time or resource usage such as memory. The compilation process itself can be done before program execution (static compilation), or during program execution (dynamic compilation). This large design space is the main challenge for compiler writers. Many tradeoffs have to be considered in order to justify the development and implementation of a particular optimization pass or strategy. However, every compiler optimization needs to address the following three issues:

1. opportunity: When can the optimization be applied?

2. safety: Does the optimization preserve program semantics?
3. profitability: When applied, how much performance improvement can be expected?

Clearly, every program transformation should be safe. Compiler writers will be out of their jobs if safety can be ignored. Profitability has to consider any overheads introduced by an optimization, in particular runtime overheads. The combination of opportunity and profitability allows the assessment of the expected overall effectiveness of an optimization.

In principle, hardware and OS based program improvement strategies face the same challenges as compiler optimizations. However, the tradeoff decisions are different based on the acceptable cost of an optimization and the availability of information about dynamic program behavior. Hardware and OS techniques are performed at runtime where more accurate knowledge about control flow and program values may be available. Opportunity, safety and profitability checks result in execution time overheads, and therefore need to be rather inexpensive. Profitability analyses typically use a limited window of past program behavior to predict future behavior. In contrast, in a static compiler, most of the opportunity, safety and profitability checks are done at compiler time, i.e., not at program execution time, allowing more aggressive program transformations in terms of affected scope and required analyses. Since the entire program is available to the compiler, future program behavior may be predicted more accurately in the cases where static analysis techniques are effective. Purely static compilers do not perform well in cases where program behavior depends on dynamic values that cannot be determined or approximated at compile time. However, in many cases, the necessary dynamic information can be derived at compile time or code optimization alternatives are limited, allowing the appropriate alternative to be selected at runtime based on compiler generated tests. The ability of the compiler to reshape program behavior through aggressive whole program analyses and transformations that is a key advantage over hardware and OS techniques, exposing optimization opportunities that were not available before. In addition, aggressive whole program analyses allow optimizations with high runtime overheads which typically require a larger scope in order to assess their profitability.

In the following, several promising compiler optimization techniques are discussed, together with an assessment of their potential benefits. These optimizations include remote task mapping, resource hibernation, and dynamic voltage and frequency scaling.

35.3 Power vs. Energy vs. Performance

Optimizing compilers need underlying performance models and metrics to be able to transform the program code for a specific optimization goal. These models and metrics guide the compiler to make selections among program transformation alternatives. If one optimization goal subsumes another, there is no need to develop separate models and metrics for the subsumed models. In this section we address the question whether power, energy, and performance should be considered separate compiler optimization goals or not.

Power vs. Energy

Optimizing for minimal power dissipation or minimal energy usage may have different metrics, and therefore result in different optimization strategies. One possible metric for power and energy is that of *activity level* at any given point during program execution and *total amount of activities* for a program region, respectively. The more "work" is done at a program point, the more power is dissipated. Given these metrics, is optimizing for power the same as optimizing for energy? The answer will depend on the particular definition of "work".

An optimizing compiler may define work as the number of instructions executed at a given point in time. This model assumes that (1) a fixed amount of power is associated with each executed instruction, and that (2) the power dissipation of an instruction is independent of its particular operand values or other executing instructions. Figure 1 illustrates this case. By reordering or rescheduling instructions, for instance in a VLIW or superscalar architecture, the initial power profile of a program region as shown on the left of Figure 1 may ideally be transformed into the one shown on the right. While the peak power dissipation is different for both profiles, the energy usage is the same. In other words, activity or work rescheduling can be an effective way to reduce peak power dissipation while having no impact on energy usage. Therefore, peak power reduction may be an optimization objective different from energy reduction.

For power models based on bit-level switching activities as its work notion, rescheduling instructions may also target overall energy usage by grouping instructions based on their particular bit patterns. In addition to instruction scheduling, a careful selection of register names in the code generation phase of a compiler can result in code sequences that have bit patterns with less switching activities, for instance due to the reuse of "similar" register names [7].

Due to the particular chemical characteristics of some batteries, highly varying discharge rates, i.e., varying power dissipations, may reduce the lifetime of a battery significantly. By "smoothing" the power dissipation profile of an application through instruction scheduling and reordering, the usable energy of a battery can be significantly increased [11].

From now on, we will not distinguish between the optimization objectives of reducing peak power dissipation and overall energy usage unless explicitly stated.

Power/Energy vs. Performance

Early work on optimizing compilers for power and energy management suggested that optimization transformations for performance subsume those for power and energy management. Therefore, power/energy is not an optimization objective

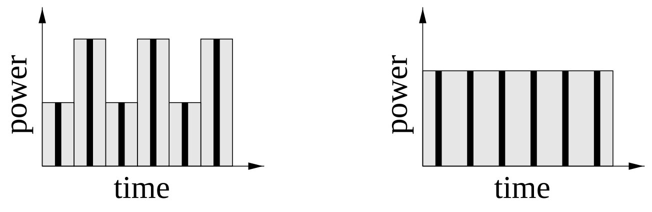

Figure 1: Optimizing for power vs. energy: Two possible power profiles of an example program region.

in its own right [13]. Traditional optimizations such as common subexpression elimination, partial redundancy elimination, strenght reduction, or dead code elimination increase the performance of a program by reducing the work to be done during program execution [12, 2]. Clearly, reducing the workload may also result in power/energy savings. Memory hierarchy optimizations such as loop tiling and register allocation try to keep data closer to the processor since such data can be faster accessed. Keeping a value in an on-chip cache instead of an off-chip memory, or in a register instead of the cache also saves power/energy due to reduced switching activities and switching capacitance.

However, there is fundamental difference in the models and metrics used for performance and those for power/energy optimizations. Many performance models have the notion of a critical path, i.e., a sequence of instructions or activities that will dominate the overall program execution time. If an optimization introduces activities on the non-critical path, performance is not affected. Therefore, as long as these non-critical activities lead to an overall decrease of the critical path (at least in most cases), the optimization is beneficial. In the context of power/energy optimizations, this is not true. Any activity, whether on or off the critical path will contribute to the overall power dissipation and energy usage.

Figure 2 shows an example that illustrates the differences in optimizing for power/energy versus optimizing for performance for a source-level transformation, in this case loop invariant code motion [12, 2]. In the example program, the assignment `a = b * 2` is assumed to be loop invariant. For a traditional scalar architecture, loop invariant code motion will move the assignment out of the loop, resulting in the code on the right of Figure 2. In a VLIW architecture, the code on the left may be best if empty VLIW instruction slots are available to execute the loop invariant assignment for each iteration of the loop. Although the assignment is done ten times, it may reduce the overall critical path. Depending on the particular overall compilation strategy used, moving the assignment out of the loop may actually increase the critical path. In the context of power/energy optimization, performing redundant computations should be avoided, and therefore moving the invariant assignment out of

```
for (i=0; i<10; i++) {            a = b * 2;
    a = b * 2;                    for (i=0; i<10; i++) {
    c[i] = d[i] + 2.0;                c[i] = d[i] + 2.0;
}                                 }
```

Figure 2: Example code fragment to illustrate power vs. performance optimization strategies.

---

the loop typically leads to power and energy savings.

Another example where optimizations for power/energy may be different from that for performance is speculative execution. Speculation performs activities "ahead of time" based on some assumptions about the future behavior of the program. If these assumptions turn out to be false, additional work may be necessary to undo the impact of the speculative performed activities. Software prefetching is an example of such a transformation. The compiler may insert prefetch instructions for memory accesses across control branches. Assuming that the target machine allows multiple outstanding loads, this optimization can be very effective. Again, as long as the speculative activity can be hidden on the non-critical execution path, no negative impact on performance will occur. In the context of power/energy optimizations every additional, speculative activity has to be compensated for by the overall power/energy benefit of the optimization in order to make things not worse. In other words, the window of profitability has to be larger for power/energy optimizations than performance optimizations. This does not mean that speculation cannot be applied for power/energy optimizations, but suggests a less aggressive application of such a transformation by restricting it to the cases where the benefit is likely.

Summary

In recent years, reducing power dissipation and energy consumption of a program have become optimization goals in their own right, no longer considered a by-product of traditional performance optimizations which mainly try to reduce program execution times. Power and energy optimizations can be implemented in hardware through circuit design, by the operating system through scheduling techniques that consider the power and energy requirements of active processes, and by the compiler through compile-time analyses, code reshaping, and hints to the operating system. The following issues should be considered during the design of an optimizing compiler for power/energy management:

1. You can run but you cannot hide: All instructions, including instructions on the non-critical path contribute to the overall power dissipation and energy consumption. As a result, power/energy optimizations have a higher threshold for profitability than performance optimization if they require additional instructions to be executed.

2. Keep the overall picture in mind: A power/energy optimization with a slight performance penalty may be profitable for a single system component (e.g.: cache, CPU, memory), it may not be profitable for the overall system due to its impact on the power/energy requirements of other system components. In addition, the power/energy characteristics of other active processes have to be considered in a multi-programming environment.

3. You cannot beat hardware: If an operation is implemented in hardware, and an application can take advantage of this hardware (e.g.: floating point unit), a compiler should try to generate code for it. If the hardware dissipates power while idle, the compiler needs to be able to disable it during such idle periods.

35.4 List of Optimizations

In the following, three compiler optimizations are discussed. These optimizations are just examples, and are presented to illustrate the potential benefits of compile time power/energy management. This list is by no means complete.

Dynamic Voltage and Frequency Scaling

Dynamic voltage scaling (DVS) is recognized as one of the most effective power reduction techniques. It exploits the fact that a major portion of power of CMOS circuitry scales quadratically with the supply voltage [3]. As a result, lowering the supply voltage can significantly reduce power dissipation. For non-interactive applications such as movie playing, decompression, and encryption, fast processors reduce device idle times, which in turn reduce the opportunities for power savings through hibernation strategies. In contrast, DVS techniques are still beneficial in such cases, i.e., DVS reduces power even when these devices are active. However, DVS comes at the cost of performance degradation. An effective DVS algorithm is one that intelligently determines *when* to adjust the current frequency-voltage setting (*scaling points*) and to *which* frequency-voltage setting (*scaling factors*), so that considerable savings in energy can be achieved while the required performance is still delivered.

One possible compiler-directed algorithm identifies program regions where the CPU can be slowed down with negligible performance loss [6]. It is implemented as a source-to-source level transformation using the SUIF2 [1] compiler infrastructure. Physical measurements on a laptop with a 600-1200 MHz AMD Athlon 4 processor show that total system energy savings of up to 23% can be achieved with performance degradation of less than 5% for the `SPECfp95` benchmarks. On average, the energy and energy-delay product are reduced by 11% and 9%, respectively, at the cost of the performance slowdown of 2%. It was also discovered that the energy usage of the programs using this DVS algorithm is within 6% from the theoretical lower bound.

Resource Hibernation

A common approaches to increase energy efficiency puts idle resources or entire devices in low-power (hibernation) states until they have to be accessed again. The transition to a lower power state usually occurs after a period of inactivity (an *inactivity threshold*), and the transition back to active state usually occurs on demand. Unfortunately, the transitions to and from the low-power state can consume significant time and energy. Nevertheless, this strategy works well when there is enough idle time to justify incurring such costs.

Source-level transformations can be used to reshape the program behavior such that inactivity thresholds of a device or component are extended, allow hibernation to be more effective. By allowing the compiler to give hints to the operating system about expected idle times of these components and devices, the operating system is able to issue deactivation directives earlier and activation directives just in time before the device or component is used again. In addition, the operating system can use these hints to implement the most efficient policy for the set of active processes. The results reported in [5] show that on a set of streamed and non-streamed application, the reshaped programs can achieve disk energy reductions ranging from 55% to 89% (70% on average) under a sophisticated energy management policy with only a small performance degradation.

Remote Task Mapping

Mobile devices come in many flavors, including laptop computers, Webphones, pocket computers, Personal Digital Assistance (PDAs), and intelligent sensors. Many such devices already have wireless communication capabilities, and we expect most future systems to have such capabilities. There are two main differences between mobile and desk-top computing systems, namely the source of the power supply and the amount of available resources. Mobile systems operate entirely on battery power most or all the time. The resources available on a mobile system can be expected to be at least one order of magnitude less than those of a "wall-powered" desk-top system with similar technology. This fact is mostly due to space, weight, and power limitations placed on mobile platforms. Such resources include the amount and speed of the processor, memory, secondary storage, and I/O. With the development of new and even more power-hungry technology, we expect this gap to widen even more. Remote task mapping is a technique that tries to off-load computation to a remote server, thereby saving power and energy on the mobile devices [8, 10].

A possible compilation strategy that generates two versions of the initial application, one to be executed on the mobile device (client), and the other on a machine connected to the mobile device via a wireless network (server) [8]. The client and server codes have to be able to deal with disconnection events. The proposed compilation strategy uses checkpointing techniques to allow the client to monitor program progress on the server, and to request checkpoint data in order to reduce the performance penalty in case of a possible server and/or network failure.

The reported results have been obtained by actual power measurements of an image processing application (face detection and face recognition) on three client systems, (1) the StrongARM based low-power SKIFF system developed at Compaq's Cambridge Research Laboratory, (2) Compaq's commercially available StrongARM based iPAQ H3600, and (3) a PentiumII based laptop. Initial experiments show that energy consumption can be reduced significantly, in some cases up to one order of magnitude, depending on the selected characteristics of the mobile device, remote host, and wireless network.

35.5 Future Compiler Research for Power/Energy

Compiler research for power and energy management is still in its infancy. Such research requires platforms that expose power and energy management features to higher software levels such as the compiler through standardized interfaces (APIs). While efforts have been made in some areas (e.g.: ACPI [4]), more work needs to be done.

In addition, the lack of a reliable and effective evaluation infrastructures for power and energy optimizations has significantly hampered compiler research. The compiler community relies mostly on physical measurements on existing target systems for a set of representative benchmarks to evaluate the benefits of a given optimization or set of optimizations. Simulation results are accepted as an indication of a potential benefit of an optimization, but are typically not considered sufficient proof that the optimization is worthwhile in practice. What is needed is an evaluation infrastructure for power and energy optimizations that consists of a combination of physical measurements and performance modeling. Physical measurements need to include current and voltage measurements, as well as temperature measurements. Performance models are needed for the CPU, memory subsystems, controllers, communication modules, and I/O devices such as the disk and screen. This technology is crucial to be able to understand and assess the benefits of a proposed optimization for the entire target system, subsets of system components, or single system components.

Acknowledgement

This work has been partially supported by NSF CAREER award #9985050. Any opinions and conclusions expressed in this chapter are those of the author, and do not necessarily reflect the view of the National Science Foundation.

References

[1] National Compiler Infrastructure (NCI) project. Overview available online at http://www-suif.stanford.edu/suif/nci/index.html., Co-funded by NSF/DARPA, 1998.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, MA, second edition, 1986.

[3] T. Burd and R. Brodersen. Energy efficient CMOS microprocessor design. In *the 28th Hawaii International Conference on System Sciences (HICSS-95)*, January 1995.

[4] Intel Corp., Microsoft Corp., and Toshiba Corp. ACPI implementers' guide. Draft, February 1998.

[5] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, Charlottesville, VA, September 2002.

[6] C-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *ACM SIGPLAN Conference on Programming Languages, Design, andImplementation (PLDI'03)*, San Diego, CA, June 2003.

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

[8] U. Kremer, J. Hicks, and J. Rehg. A compilation framework for power and energy management on mobile computers. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, Cumberland, KT, August 2001.

[9] R. Leupers. Compiler design issues for embedded processors. *IEEE Design & Test of Computers*, 19(4):51–58, July/August 2002.

[10] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2001)*, Atlanta, GA, November 2001.

[11] T. Martin and D. Siewiorek. The impact of battery capacity and memory bandwidth on CPU speed-setting: A case study. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 200–205, San Diego, CA, August 1999.

[12] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Franscisco, CA, 1997.

[13] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2/3):1–18, 1996.