Embedded Tutorial 2: Compilers for Power and Energy Management

Ulrich Kremer
Department of Computer Science
Rutgers University
New Jersey, USA

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.

Recently, power dissipation and energy consumption have become optimization goals in their own right, no longer being considered a by-product of traditional performance optimizations. Effective power and energy management is important to prolong battery life and to reduce heat dissipation. Developing compile-time techniques for application specific power and energy management is an exciting new challenge. In this tutorial, I will give an overview of current approaches to compiler-directed power and energy mangement. I will discuss several promising compiler optimization techniques in detail, together with an assessment of their potential benefits. These optimizations include remote task mapping, resource hibernation, dynamic voltage and frequency scaling, and quality of result tradeoffs. Based on preliminary experiences with these optimizations, I will present a compiler writer's wish list for hardware architects and OS designers in order to support application specific power and energy management. An overview of future challenges will conclude the tutorial.

Copyright is held by the author/owner.
ISLPED'02, August 12-14, 2002, Monterey, California, USA.
ACM 1-58113-475-4/02/0008.