Addressing Fundamental Bottlenecks In Link-Time and Dynamic Optimization

Chris Lattner, Shashank Shekhar, Anand Shukla, and Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,shekhar,ashukla,vadve}@cs.uiuc.edu

Abstract. The popularity of object-oriented programming languages, component libraries, and dynamically linked libraries all significantly limit the effectiveness of static analysis and optimization. These trends have led to significant recent interest in link-time interprocedural optimization and runtime optimization. Unfortunately, these approaches present some difficult technical challenges that have limited their use so far. One common limitation is that machine-level object code (the output of traditional source-level compilers) can be very difficult to analyze and transform. Link-time and runtime optimizers working with machine code generally cannot perform sophisticated high-level analyses and optimizations, e.g., sophisticated pointer analysis exploiting high-level type information and function signatures, data structure layout reorganization, loop transformations requiring array dependence analysis, and many others. A second fundamental limitation of runtime optimization is its runtime overhead, which remains a major limiting factor despite a long history of research. There appears little reason to believe that this fundamental trade-off can be mitigated through software alone.

In one part of our work, we have developed a compilation strategy that directly addresses the challenge of optimizing object code at both link time and runtime (and does so without exporting compiler-specific internal representations, as in some current commercial link-time systems). In particular, we propose a rich instruction set called Low Level Virtual Machine (LLVM) that uses primarily low-level operations, similar to a RISC assembly language, but provides rich information about the operands. This includes high-level, language-independent type information such as pointers, structures, arrays. It also includes primitive instructions for typed dynamic memory allocation, In addition, all operands are in static single assignment (SSA) form, which can enable many efficient optimization algorithms without the expense of computing SSA at link-time or runtime independent source-level compilers can be extended to generate LLVM object code, relatively easily (as we have done with CC).

LLVM enables a wide range of high-level interprocedural optimizations, at runtime.

Separately, we are investigating how hardware support can be used to address the second fundamental bottleneck above, by moving the overheads of dynamic compilation off the critical path of target program execution. We propose the use of a simple compilation coprocessor that generalizes two recently proposed coprocessors for profiling and for accelerating hardware optimizations. The coprocessor is designed to support both IT compilers (e.g., Fortran and C) and dynamic optimizing compilers. Aggressive runtime optimizations can exploit such hardware support. Longer term, we aim to compare other alternatives for moving runtime optimizations off the critical path (including simultaneous multi-threading, and multiprocessor-on-chip systems). More broadly, the aim of this work is to develop new runtime optimizations that can exploit any such hardware support to perform sophisticated optimizations off the program's critical path.

*This work is supported by an NSF CAREER Award, EIA-0093426, through the NSF Next Generation Software Program.