Compiler Optimization Phase Sequence Search*

Suresh Purini, IIIT-H

1 Introduction and Problem Description

Modern optimizing compilers are organized in to a sequence of 3 modules – Front End, Optimizer (Middle End) and Back End, as in Figure 1. The Front End of the compiler checks whether an input program is syntactically and semantically well-formed, if not it will generate suitable error messages for the user. Otherwise it will generate an intermediate representation (IR) of the program which the Optimizer component will optimize with respect to parameters like speed, memory footprint, power etc. The optimized IR is in turn translated into target machine code by the Back End of the compiler. It has to be noted Back End also performs certain machine dependent code optimizations on the IR which could be Back End specific.

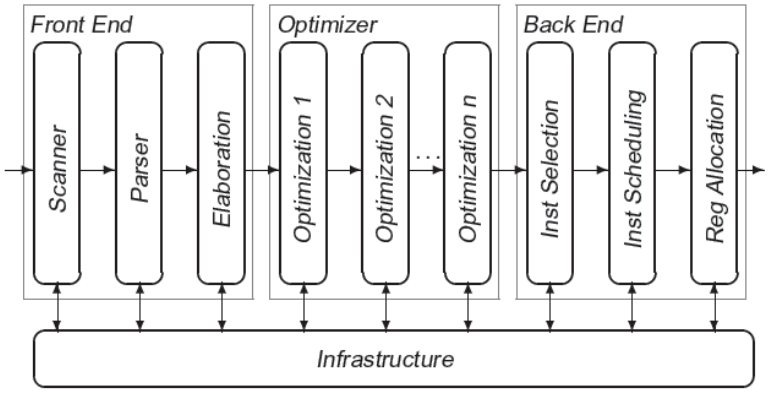

Figure 1: Structure of a Three Phase Compiler

The Optimizer component of the compiler is a organized as a collection of *passes* of two types: *analysis* and *transformation passes*. Analysis passes analyze the program and collect necessary information which the transformation passes use to optimize the IR. Typically a compiler comes with a set of program optimizations and a partial order or more generally a set of constraints on the order in which the program optimizations can be invoked. Further some optimizations like loop unrolling and loop tiling are parameterized. And there are optimizations like data prefetching for which we have both hardware and software approaches, and different possible prefetch policies. The compiler can possibly have different algorithmic implementations for modules like register allocation and instruction scheduling. The question here is how we should choose the parameters, algorithmic implementations, and in what order should we apply these compiler optimizations to

*This document is strictly for IIIT-H internal purpose as I am using certain copyrighted material in this lecture notes.

generate target code which runs fast[1]. This problem arises because a compiler optimization could enable or disable optimization opportunities for the later phases. For example it may be a good idea to invoke constant folding optimization after constant propagation optimization. Another example is the well-known phase ordering problem between instruction scheduling and register allocation[2]. The possible choice for parameters and the choice in ordering optimization passes provides for combinatorially exponentially many ways of composing a compiler to act on a program. If we call each choice of these parameters as a *optimization sequence*, then given a program for compilation we have to find the optimal optimization sequence. Compilers provide various default optimization levels (like -O2, -O3 etc.) each corresponding to a certain optimization sequence which could give decent performance on typical programs but these configurations may not be the best for them. It is customary in industry to report the performance of the compilers and processors on benchmark programs by using the best possible configurations. In general the optimal compiler configurations depends on the program, target architecture and the compiler itself[3]. We shall formalize the problem as follows. Let $\mathcal{S} = (\mathcal{M}, \mathcal{C})$ be a Compilation System with $\mathcal{M}$ being the set of available optimization modules and $\mathcal{C}$ consists of the ordering constraints on the application of these modules in a sequence. For example if an optimization pass transforms the Intermediate Representation (IR) of a program into a low level form, then all optimizations requiring high level IR cannot be applied there after. A compilation sequence $s_1 \cdots s_n \in \mathcal{M}^*$ acts on a program P in the order $s_n(s_{n-1}(\cdots(s_1(P))\cdots))$. Let $\mathcal{M}^*_{\mathcal{C}}$ contains all the valid compilation sequences. Note that it is possible for an optimization module to occur more than once in a compilation sequence. For example we may want to apply Dead Code Elmination more than once.

Optimization Orchestration Problem. *Given a compilation system $\mathcal{S} = (\mathcal{M}, \mathcal{C})$ and program P, construct an optimal compilation sequence $S \in \mathcal{M}^*_{\mathcal{C}}$.*

The above problem is not well-defined as the optimal compilation sequence for a program could depend on the *program input* and the *target architecture*[4]. This problem is also called as *phase ordering problem* in the literature. Parameterized optimization modules like loop unrolling, loop tiling, data prefetching can be handled in this problem framework by considering each parameterized version as a seperate optimization module. However the the performance impact of such an inclusive approach is not yet studied.

There are broadly two approaches adopted to solve this problem.

1. **Heuristic Search Techniques** We can apply heuristic search techniques like hill-climbing, simulated annealing, genetic algorithms etc. to search for optimal compiler configurations.

2. **Machine Learning Based Prediction Models** We can build prediction models by applying machine learning techniques.

[1]We focus on the speed of the executable code alone in this project leaving aside other performance parameters like size, power consumed etc.

[2]If we do instruction scheduling first, then the live range of the variables increases, thereby increasing the register pressure. If we do register allocation first, due to register reuse the compiler may introduce false dependencies between instructions and the instruction scheduler may thereby lose code movement opportunities

[3]When a compiler is retargeted to a new architecture, various parameters, algorithms, and analytical models guiding the algorithms in the compiler have to be fine tuned for near-optimal performance of the compiler (in terms of the code it generates). Also typical compiler configurations which works on typical workloads also have to be determined. Given the vast number of possible targets and different architectures within each target platform emerging at all levels of computing – embedded, personal computing and workstations – retargeting the compiler has become tedious both for the compiler writers to find good compiler configurations for typical workloads and also for the end users of the compilers to find the optimizal compiler configurations for their applications.

[4]However we could hope that once the target architecture is fixed, then finding the optimal optimization sequence for a program on a typical input is sufficient

LLVM
-adce, -always-inline, -argpromotion, -codegenprepare, -constmerge, -constprop, -correla ted-propagation, -dce, -deadargelim, -die, -dse, -early-cse, -globaldce, -globalopt, -gvn, -in dvars, -inline, -instcombine, -instsimplify, -internalize, -ipconstprop, -ipsccp, -jump-threading , -licm, -loop-deletion, -loop-idiom, -loop-instsimplify, -loop-reduce, -loop-rotate, -loop-simplif y, -loop-unroll, -loop-unswitch, -loops, -lower-expect, -loweratomic, -lowerinvoke, -lowerswitch, -memcpyopt, -mergefunc, -mergereturn, -partial-inliner, -prune-eh, -reassociate, -scalarrepl, -s ccp, -simplify-libcalls, -simplifycfg, -sink, -tailcallelim, -targetlibinfo, -no-aa, -tbaa, -bas icaa, -basiccg, -functionattrs, -scalarrepl-ssa, -domtree, -lazy-value-info, -lcssa, -scalar-evolu tion, -memdep, -strip-dead-prototypes
Intel C Compiler
GCC
Use the list of optimizations given under the optimization options subsection of the gcc man page.

In this project we shall focus only on *heuristic search techniques*, although you are free to use *machine learning techniques* also.

2 Project

In this project you need to come up with techniques to solve the compiler optimization orchestration problem. This is an open ended problem and you have the chance to beat the state-of-the-art techniques also. You will be working on three different compilers: LLVM, Intel C Compiler and GCC. The optimizations you have to consider for each of these compilers is listed in the Table 2. For the GCC and Intel Compiler, you need to optimization combinations and for LLVM you need to find the optimization sequences. For LLVM use the following framework.

1. Using `clang` transform the source program into llvm IR. You have to use -O0 option at this step.
2. Apply `-scalarrepl` or `-mem2reg` optimization on the llvm IR file generated in the step 1 and use the transformed llvm IR program as the base program.
3. Invoke a search procedure which applies various optimization sequences on the IR file generated in the step 2 and tests the run time of each generated IR file. You may want to invoke `llc` command at -O2 level to transform the IR file to the target machine code.

This project is all about how efficiently you can search for near optimal optimization sequences in Step 3.