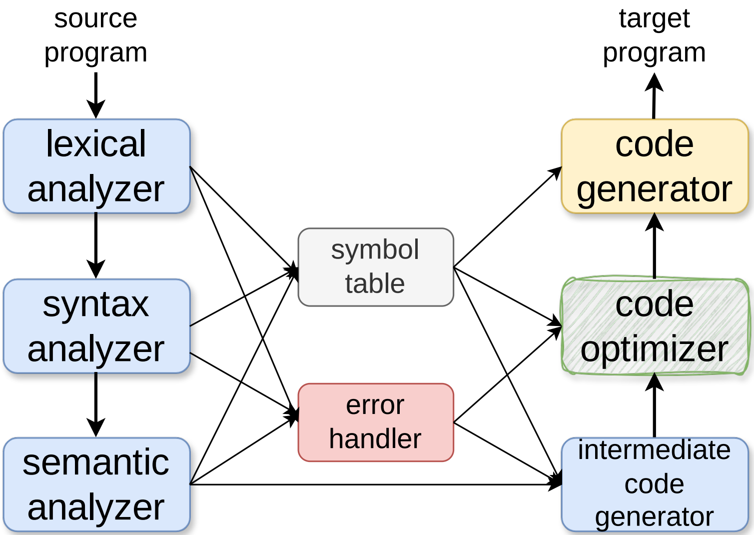
CS 335: A Brief Introduction to Compiler Optimizations

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

An Overview of Compilation



Goal of Compiler Optimizations Phase

- Intermediate code can contain many inefficiencies (e.g., repeated evaluation of sub-expressions)
- Optimizer phase **aims to improve** the performance of the input code according to some metric of interest
  - For example, run time, code size, or energy efficiency
- Maintaining semantic equivalence is important
  - The optimized version should **not** introduce a behavior that was not originally possible in the unoptimized version
    - Prohibits optimizations that affect behavior only in rare scenarios
- Two broad categories of optimizations

Machine independent	▶ Optimizations that are applicable irrespective of the target
Machine dependent	▶ Optimizations specific to a target

Scope of Optimizations

Local
- Analysis is restricted to **a BB** (control flow information is not used)
- Not all optimizations can be applied locally (e.g., loop optimizations)

Global
- Scope includes **multiple BBs** but is restricted to within functions
  - Could be restricted to a smaller scope, e.g. a loop
- Most compilers implement global optimizations using techniques like dataflow analysis

Inter-procedural
- Scope includes **multiple functions** (possibly files)
- Challenging to implement (e.g., analyze various parameter passing mechanisms)

Whole program
- All the required definitions are available to the compiler/linker
- Enables aggressive optimizations like cross-module inlining
- Also known as link-time optimizations (LTO)

Optimizing Compiler

- An **optimizing** compiler focuses more on the middle-end compared to mainstream compilers
- Most complexity in modern compilers is in the optimizer
  - Usually contributes most to the compile time and is the largest in terms of lines of code

Machine-Independent Optimizations

Optimizations in LLVM

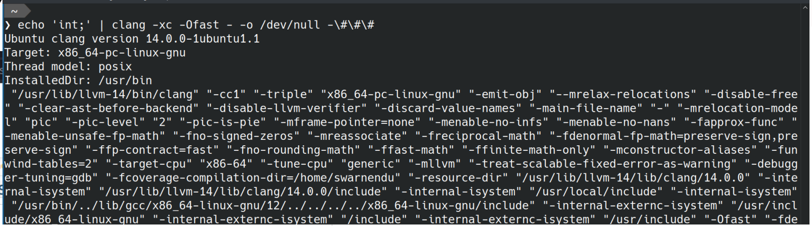
https://llvm.org/docs/Passes

LLVM COMPILER INFRASTRUCTURE

LLVM Home | Documentation » User Guides » LLVM's Analysis an

LLVM's Analysis and Transform Pas

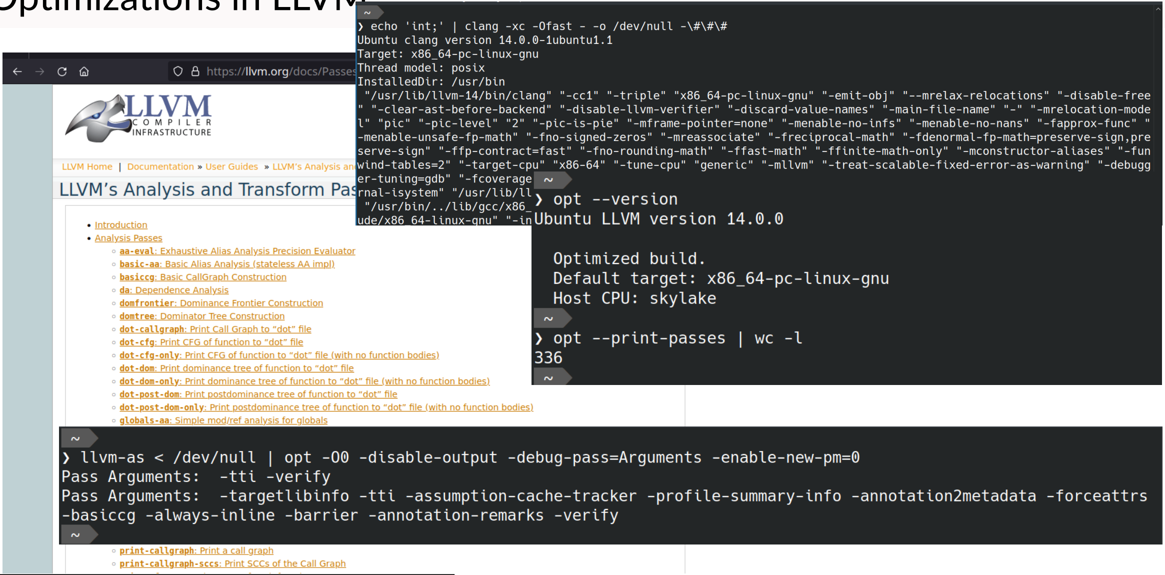
- Introduction
- Analysis Passes
  - aa-eval: Exhaustive Alias Analysis Precision Evaluator
  - basic-aa: Basic Alias Analysis (stateless AA impl)
  - basiccg: Basic CallGraph Construction
  - da: Dependence Analysis
  - domfrontier: Dominance Frontier Construction
  - domtree: Dominator Tree Construction
  - dot-callgraph: Print Call Graph to "dot" file
  - dot-cfg: Print CFG of function to "dot" file
  - dot-cfg-only: Print CFG of function to "dot" file (with no function bodies)
  - dot-dom: Print dominance tree of function to "dot" file
  - dot-dom-only: Print dominance tree of function to "dot" file (with no function bodies)
  - dot-post-dom: Print postdominance tree of function to "dot" file
  - dot-post-dom-only: Print postdominance tree of function to "dot" file (with no function bodies)
  - globals-aa: Simple mod/ref analysis for globals
  - instcount: Counts the various types of Instructions
  - iv-users: Induction Variable Users
  - lazy-value-info: Lazy Value Information Analysis
  - lint: Statically lint-checks LLVM IR
  - loops: Natural Loop Information
  - memdep: Memory Dependence Analysis
  - print<module-debuginfo>: Decodes module-level debug info
  - postdomtree: Post-Dominator Tree Construction
  - print-alias-sets: Alias Set Printer
  - print-callgraph: Print a call graph
  - print-callgraph-sccs: Print SCCs of the Call Graph



```
~
❯ echo 'int;' | clang -xc -Ofast - -o /dev/null -\#\#\#
Ubuntu clang version 14.0.0-1ubuntu1.1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
 "/usr/lib/llvm-14/bin/clang" "-cc1" "-triple" "x86_64-pc-linux-gnu" "-emit-obj" "--mrelax-relocations" "-disable-free
" "-clear-ast-before-backend" "-disable-llvm-verifier" "-discard-value-names" "-main-file-name" "-" "-mrelocation-mode
l" "pic" "-pic-level" "2" "-pic-is-pie" "-mframe-pointer=none" "-menable-no-infs" "-menable-no-nans" "-fapprox-func" "
-menable-unsafe-fp-math" "-fno-signed-zeros" "-mreassociate" "-freciprocal-math" "-fdenormal-fp-math=preserve-sign,pre
serve-sign" "-ffp-contract=fast" "-fno-rounding-math" "-ffast-math" "-ffinite-math-only" "-mconstructor-aliases" "-fun
wind-tables=2" "-target-cpu" "x86-64" "-tune-cpu" "generic" "-mllvm" "-treat-scalable-fixed-error-as-warning" "-debugg
er-tuning=gdb" "-fcoverage-compilation-dir=/home/swarnendu" "-resource-dir" "/usr/lib/llvm-14/lib/clang/14.0.0" "-inte
rnal-isystem" "/usr/lib/llvm-14/lib/clang/14.0.0/include" "-internal-isystem" "/usr/local/include" "-internal-isystem"
 "/usr/bin/../lib/gcc/x86_64-linux-gnu/12/../../../../x86_64-linux-gnu/include" "-internal-externc-isystem" "/usr/incl
ude/x86_64-linux-gnu" "-internal-externc-isystem" "/include" "-internal-externc-isystem" "/usr/include" "-Ofast" "-fde
```

LLVM's Analysis and Transformation Passes

Optimizations in LLVM



LLVM's Analysis and Transformation Passes

Optimization Levels in LLVM

- O0 no optimizations
- O1 enables common optimizations (e.g., `-licm` and `-tailcallelim`)
- O2 extends O1 to include additional optimizations (e.g., `-gvn` and `-slp-vectorization`)
- Os builds on O2 to reduce code size
- Oz extends Os to reduce code size even further
- O3 enables optimizations that may slow down compilation or increase code size
- O4 enables whole program optimization at link-time

Constant Folding and Constant Propagation

- **Constant Folding**
  - Expressions with constant operands can be evaluated at compile time
- **Constant Propagation**
  - If the value of a variable is known to be a constant, the compiler will replace its use with that constant
  - May result in the application of constant folding

```
int x = 8 * 10 * 8;
int z = a[x];
```

```
int z = a[640];
```

Constant folding

Copy Propagation and Common Subexpression Elimination

- **Copy Propagation**
  - Replace the use of a variable with another variable, if they are guaranteed to have the same value
  - May result in common sub-expressions and redundant stores
- **Common Sub-Expression Elimination**
  - Reuse the value of a common sub-expression if it was already previously computed, and the values of the operands have not changed since
    - Useful in optimizing array index computations

```
y = x;
z = 3 + y;

i = p + q + 1;
j = p + q;
```

```
z = 3 + x;


t1 = p + q;
i = t1 + 1;
j = t1;
```

Copy propagation

Common subexpression elimination

Loop-Invariant Code Motion

Move loop-invariant code out of the loop body

```
int i = 0;
while (i < n) {
  x = y + z;
  a[i] = 6 * i + x * x;
  ++i;
}
```

```
int i = 0;
if (i < n) {
  x = y + z;
  const int t1 = x * x;
  while (i < n) {
    a[i] = 6 * i + t1;
    ++i;
  }
}
```

Loop-invariant code motion

Dead Code Elimination

Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated

```
int global;
void f() {
  int i;
  i = 1;          // dead store
  global = 1; // dead store
  global = 2;
  return;
  global = 3; // unreachable
}
```

```
int global;
void f() {
  global = 2;
  return;
}
```

Dead-code elimination

Function Inlining

- Replace a function call site with the body of the called function
  - Avoids the control transfer overhead (e.g., precall, callee prologue, callee epilogue, postcall) for frequently-executed functions
- Enables more optimizations on a larger code snippet without the need for inter-procedural analysis
- Too much inlining can hurt performance and increases the memory overhead (due to code duplication)
  - Compilers and runtimes use heuristics based on the size of the callee

Inline expansion

Loop Transformations

- Loops are one of the most commonly used constructs in HPC program
- Compiler performs many loop optimization techniques automatically
  - Examples: unrolling, permutation, reversal, fission, fusion, skewing, and tiling
- In some cases, source code modifications can enhance the optimizer's ability to transform code

D. Bacon et al. Compiler Transformations for High-Performance Computing. ACM Computing Surveys, 1994.

Loop Permutation (or Interchange)

Switch the nesting order of loops in a perfect loop nest

- Can increase parallelism, can improve spatial locality

```
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    A[i][j+1] = A[i][j] + 100;
  }
}
```

```
for (j = 0; j < M; j++) {
  for (i = 0; i < N; i++) {
    A[i][j+1] = A[i][j] + 100;
  }
}
```

```
for (i = 1; i < N; i++) {
  for (j = 1; j < N; j++) {
    C[i][j] = C[i-1][j+1];
  }
}
```

```
for (j = 1; j < N; j++) {
  for (i = 1; i < N; i++) {
    C[i][j] = C[i-1][j+1];
  }
}
```

Loop Distribution (or Fission)

```
     for (i = 1; i < N; i++) {
       for (j = 1; j < N; j++) {
S1:      A[i][j] = B[i][j] + C[i][j];
S2:      D[i][j] = A[i][j-1] * 2.0;
       }
     }
```

```
     for (i = 1; i < N; i++) {
       for (j = 1; j < N; j++) {
S1:      A[i][j] = B[i][j] + C[i][j];
       }
     }

     for (i = 1; i < N; i++) {
       for (j = 1; j < N; j++) {
S2:      D[i][j] = A[i][j-1] * 2.0;
       }
     }
```

eliminates loop-carried dependences

Challenges and Tradeoffs in Optimizations

Challenges in Effective Compiler Optimizations

- Optimizations are generally both compute- and memory-intensive
- Tradeoff in terms of a tolerable compilation time and the extent of optimizations that a compiler might provide
- Language features may complicate effective optimizations (e.g., memory aliasing and procedure with side effects)
- "Premature optimization is the root of all evil"
  - Knowing which parts of a program to optimize
  - Need a reasonably accurate upper-bound estimate of the performance
  - Correctness and security can be compromised if the program has undefined behavior

Compiler implementations have bugs too!

R. Hyde. The Fallacy of Premature Optimization. ACM Ubiquity, 2009.

X. Yang et al. Finding and Understanding Bugs in C Compilers. PLDI'11.

C. Sun et al. Toward Understanding Compiler Bugs in GCC and LLVM. ISSTA'16.

Can A Compiler Do Anything Wrong?

```
X *x = NULL;
bool done = false;
```

Thread 1

```
x = new X();
done = true;
```

Thread 2

```
while (!done) {}
x->func();
```

How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories

The Golden Age of Compilers

in an era of Hardware/Software co-design

International Conference on
Architectural Support for Programming Languages and
Operating Systems (ASPLOS 2021)

Chris Lattner
SiFive, Inc.

April 19, 2021