B4M36ESW: Efficient software
Lecture 4: C program compilation and execution

Michal Sojka
michal.sojka@cvut.cz

March 26, 2018
1. Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2. Linker

3. Execution
Outline

1 Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2 Linker

3 Execution
C/C++ compilation

Compilation phases:

1. Preprocessor
2. Parsing
3. High-level optimizations
4. Low-level optimizations
5. Linking

Open-source compilers:
- GCC
- LLVM/clang

LLVM has easier to understand the code base. GCC improves code readability as well.
C/C++ compilation

Compilation phases:

1. Preprocessor
2. Parsing
3. High-level optimizations
4. Low-level optimizations
5. Linking

Open-source compilers:

- GCC
- LLVM/clang

LLVM has easier to understand the code base. GCC improves code readability as well.
example.c:

```c
unsigned square(unsigned x) {
    unsigned sum = 0, tmp;
    for (unsigned i = 1; i < x; i++) {
        tmp = x;
        sum += x;
    }
    return sum + tmp;
}
```

`clang -Xclang -ast-dump -fsyntax-only example.c`

```
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
  `~FunctionDecl <example.c:1, line:9:1 line:1:10 square 'unsigned int (unsigned int)'`
    `-ParmVarDecl <col:17, col:26 col:26 used x 'unsigned int'
      `-CompoundDecl <line:2:1, line:9:1>
        `~DeclStmt <line:3:3, col:24>
          | `-VarDecl <col:3, col:18 col:18 used sum 'unsigned int' cinit
          |   `~ImplicitCastExpr <col:18> 'unsigned int' <IntegralCast>
          |    `-IntegerLiteral <col:18> 'int' 0
          `-VarDecl <col:3, col:21> col:21 used tmp 'unsigned int'
            `-ForStmt <line:4:3, line:7:3>
              | `-DeclStmt <line:4:8, col:22>
              |   | `-VarDecl <col:8, col:21 col:17 used i 'unsigned int' cinit
              |   |   `~ImplicitCastExpr <col:21> 'unsigned int' <IntegralCast>
              |   |    `-IntegerLiteral <col:21> 'int' 1
              |   |   |--<<NULL>>
              |   |   | `-BinaryOperator <col:24, col:28> 'int' '<'
              |   |   |   `-ImplicitCastExpr <col:24> 'unsigned int' <LValueToRValue>
              |   |   |   `~DeclRefExpr <col:24> 'unsigned int' lvalue Var 'i' 'unsigned int'
              |   |   |   `~ImplicitCastExpr <col:28> 'unsigned int' <LValueToRValue>
              |   |   `~DeclRefExpr <col:28> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
              |   |   `-UnaryOperator <col:31, col:32> 'unsigned int' postfix '+'
              |   |   `~DeclRefExpr <col:31> 'unsigned int' lvalue Var 'i' 'unsigned int'
              |   `-CompoundStmt <col:36, line:7:3>
              |     | `-BinaryOperator <line:5:5, col:11> 'unsigned int' '='
              |     |   `-DeclRefExpr <col:5> 'unsigned int' lvalue Var 'tmp' 'unsigned int'
              |     |   `~ImplicitCastExpr <col:11> 'unsigned int' <LValueToRValue>
              |     |   `~DeclRefExpr <col:11> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
              |     `-CompoundAssignOperator <line:6:5, col:12> 'unsigned int' '+'= ComputeLHSTy='unsigned int' Com
              |     `~DeclRefExpr <col:5> 'unsigned int' lvalue Var 'sum' 'unsigned int'
              |    `~ImplicitCastExpr <col:12> 'unsigned int' <LValueToRValue>
              |    `~DeclRefExpr <col:12> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
              |   `-ReturnStmt <line:8:3, col:16>
              |     `~BinaryOperator <col:10, col:16> 'unsigned int' '+'
              |     | `-ImplicitCastExpr <col:10> 'unsigned int' <LValueToRValue>
              |     | `~DeclRefExpr <col:10> 'unsigned int' lvalue Var 'sum' 'unsigned int'
              |     | `~ImplicitCastExpr <col:16> 'unsigned int' <LValueToRValue>
              |     | `~DeclRefExpr <col:16> 'unsigned int' lvalue Var 'tmp' 'unsigned int'
              |`-ReturnStmt <line:8:3, col:16>
```
Intermediate representation (IR)

example.c:

```c
unsigned square(unsigned x) {
    return x*x;
}
```

LLVM intermediate representation

```assembly
define i32 @square(i32) #0 {
    %2 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    %3 = load i32, i32* %2, align 4
    %4 = load i32, i32* %2, align 4
    %5 = mul i32 %3, %4
    ret i32 %5
}
```

clang -Xclang -ast-dump -fsyntax-only example.c

TranslationUnitDecl <<invalid sloc>> <invalid sloc> `FunctionDecl <example.c:1:1, line:4:1> line:1:10 square 'unsigned int (unsigned int)'` | `ParmVarDecl <col:17, col:26> col:26 used x 'unsigned int'
CompoundStmt <line:2:1, line:4:1> `ReturnStmt <line:3:3, col:12>`
  `BinaryOperator <col:10, col:12> 'unsigned int' '*'
  `ImplicitCastExpr <col:10> 'unsigned int' <LValueToRValue>`
  `DeclRefExpr <col:10> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'
  `ImplicitCastExpr <col:12> 'unsigned int' <LValueToRValue>`
  `DeclRefExpr <col:12> 'unsigned int' lvalue ParmVar 'x' 'unsigned int'`
Intermediate representation vs. assembler

example.c:

```c
unsigned square(unsigned x) {
    return x*x;
}
```

$ clang -S -emit-llvm example.c

```ll
; ModuleID = 'example.c'
source_filename = "example.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind uwtable
define i32 @square(i32) #0 {%2 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    %3 = load i32, i32* %2, align 4
    %4 = load i32, i32* %2, align 4
    %5 = mul i32 %3, %4
    ret i32 %5
}
```

attributes #0 = { noinline nounwind uwtable "correctly-round..."

!llvm.ident = !{{!0}

!0 = !{{"clang version 4.0.1-10 (tags/RELEASE_401/final)"}

$ llc -O0 -march=x86-64 example.ll

```asm
sub    sp, sp, #8
mov    r1, r0
str    r0, [sp, #4]
ldr    r0, [sp, #4]
mul    r2, r0, r0
mov    r0, r2
str    r1, [sp]
add    sp, sp, #8
mov    pc, lr
```

$ llc -O0 -march=arm example.ll

```asm
sub    sp, sp, #8
mov    r1, r0
str    r0, [sp, #4]
ldr    r0, [sp, #4]
mul    r2, r0, r0
mov    r0, r2
str    r1, [sp]
add    sp, sp, #8
mov    pc, lr
```

Assembler generation from IR is detailed later.

IR is machine independent
Compiler » High-level optimizations

Outline

1. Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2. Linker

3. Execution
Optimizations in general

- Many, many options
- gcc -Q --help=optimizers -O2
High-level optimizations

Analysis passes – add information for use in other passes

- Exhaustive Alias Analysis Precision Evaluator (-aa-eval)
- Basic Alias Analysis (stateless AA impl) (-basicaa)
- Basic CallGraph Construction (-basiccg)
- Count Alias Analysis Query Responses (-count-aa)
- Dependence Analysis (-da)
- AA use debugger (-debug-aa)
- Dominance Frontier Construction (-domfrontier)
- Dominator Tree Construction (-domtree)
- Simple mod/ref analysis for globals (-globalsmodref-aa)
- Counts the various types of Instructions (-instcount)
- Interval Partition Construction (-intervals)
- Induction Variable Users (-iv-users)
- Lazy Value Information Analysis (-lazy-value-info)
- LibCall Alias Analysis (-libcall-aa)
- Statically lint-checks LLVM IR (-lint)
- Natural Loop Information (-loops)
- Memory Dependence Analysis (-memdep)
- Decodes module-level debug info (-module-debuginfo)
- Post-Dominance Frontier Construction (-postdomfrontier)
- Post-Dominator Tree Construction (-postdomtree)
- Detect single entry single exit regions (-regions)
- Scalar Evolution Analysis (-scalar-evolution)
- ScalarEvolution-based Alias Analysis (-scev-aa)
- Target Data Layout (-targetdata)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowserswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
# High-level optimizations (clang)

## Transform passes

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggressive Dead Code Elimination</td>
<td><code>(-adce)</code></td>
</tr>
<tr>
<td>Inliner for always_inline functions</td>
<td><code>(-always-inline)</code></td>
</tr>
<tr>
<td>Promote ‘by reference’ arguments to scalars</td>
<td><code>(-argpromotion)</code></td>
</tr>
<tr>
<td>Basic-Block Vectorization</td>
<td><code>(-bb-vectorize)</code></td>
</tr>
<tr>
<td>Profile Guided Basic Block Placement</td>
<td><code>(-block-placement)</code></td>
</tr>
<tr>
<td>Break critical edges in CFG</td>
<td><code>(-break-crit-edges)</code></td>
</tr>
<tr>
<td>Optimize for code generation</td>
<td><code>(-codegenprepare)</code></td>
</tr>
<tr>
<td>Merge Duplicate Global Constants</td>
<td><code>(-constmerge)</code></td>
</tr>
<tr>
<td>Simple constant propagation</td>
<td><code>(-constprop)</code></td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td><code>(-dce)</code></td>
</tr>
<tr>
<td>Dead Argument Elimination</td>
<td><code>(-deadargelim)</code></td>
</tr>
<tr>
<td>Dead Type Elimination</td>
<td><code>(-deadtypeelim)</code></td>
</tr>
<tr>
<td>Dead Instruction Elimination</td>
<td><code>(-die)</code></td>
</tr>
<tr>
<td>Dead Store Elimination</td>
<td><code>(-dse)</code></td>
</tr>
<tr>
<td>Deduce function attributes</td>
<td><code>(-functionattrs)</code></td>
</tr>
<tr>
<td>Dead Global Elimination</td>
<td><code>(-globaldce)</code></td>
</tr>
<tr>
<td>Global Variable Optimizer</td>
<td><code>(-globalopt)</code></td>
</tr>
<tr>
<td>Global Value Numbering</td>
<td><code>(-gvn)</code></td>
</tr>
<tr>
<td>Canonicalize Induction Variables</td>
<td><code>(-indvars)</code></td>
</tr>
<tr>
<td>Function Integration/Inlining</td>
<td><code>(-inline)</code></td>
</tr>
<tr>
<td>Combine redundant instructions</td>
<td><code>(-instcombine)</code></td>
</tr>
<tr>
<td>Internalize Global Symbols</td>
<td><code>(-internalize)</code></td>
</tr>
<tr>
<td>Interprocedural constant propagation</td>
<td><code>(-ipconstprop)</code></td>
</tr>
<tr>
<td>Interprocedural Sparse Conditional Constant Propagation</td>
<td><code>(-ipsccpp)</code></td>
</tr>
<tr>
<td>Jump Threading</td>
<td><code>(-jump-threading)</code></td>
</tr>
<tr>
<td>Loop-Closed SSA Form Pass</td>
<td><code>(-lcssa)</code></td>
</tr>
<tr>
<td>Loop Invariant Code Motion</td>
<td><code>(-licm)</code></td>
</tr>
<tr>
<td>Delete dead loops</td>
<td><code>(-loop-deletion)</code></td>
</tr>
<tr>
<td>Extract loops into new functions</td>
<td><code>(-loop-extract)</code></td>
</tr>
<tr>
<td>Extract at most one loop into a new function</td>
<td><code>(-loop-extract-single)</code></td>
</tr>
<tr>
<td>Loop Strength Reduction</td>
<td><code>(-loop-reduce)</code></td>
</tr>
<tr>
<td>Rotate Loops</td>
<td><code>(-loop-rotate)</code></td>
</tr>
<tr>
<td>Canonicalize natural loops</td>
<td><code>(-loop-simplify)</code></td>
</tr>
<tr>
<td>Unroll loops</td>
<td><code>(-loop-unroll)</code></td>
</tr>
<tr>
<td>Unswitch loops</td>
<td><code>(-loop-unswitch)</code></td>
</tr>
<tr>
<td>Lower atomic intrinsics to non-atomic form</td>
<td><code>(-loweratomic)</code></td>
</tr>
<tr>
<td>Lower invokes to calls, for unwindless code generators</td>
<td><code>(-lowerinvoke)</code></td>
</tr>
<tr>
<td>Lower SwitchInsts to branches</td>
<td><code>(-lowerswitch)</code></td>
</tr>
<tr>
<td>Promote Memory to Register</td>
<td><code>(-mem2reg)</code></td>
</tr>
<tr>
<td>MemCpy Optimization</td>
<td><code>(-memcpyopt)</code></td>
</tr>
<tr>
<td>Merge Functions</td>
<td><code>(-mergefunc)</code></td>
</tr>
<tr>
<td>Unify function exit nodes</td>
<td><code>(-mergereturn)</code></td>
</tr>
<tr>
<td>Partial Inliner</td>
<td><code>(-partial-inliner)</code></td>
</tr>
<tr>
<td>Remove unused exception handling info</td>
<td><code>(-prune-eh)</code></td>
</tr>
<tr>
<td>Reassociate expressions</td>
<td><code>(-reassociate)</code></td>
</tr>
<tr>
<td>Demote all values to stack slots</td>
<td><code>(-reg2mem)</code></td>
</tr>
<tr>
<td>Scalar Replacement of Aggregates</td>
<td><code>(-sroa)</code></td>
</tr>
<tr>
<td>Sparse Conditional Constant Propagation</td>
<td><code>(-sccp)</code></td>
</tr>
<tr>
<td>Simplify the CFG</td>
<td><code>(-simplifycfg)</code></td>
</tr>
<tr>
<td>Code sinking</td>
<td><code>(-sink)</code></td>
</tr>
<tr>
<td>Strip all symbols from a module</td>
<td><code>(-strip)</code></td>
</tr>
<tr>
<td>Strip debug info for unused symbols</td>
<td><code>(-strip-dead-debug-info)</code></td>
</tr>
<tr>
<td>Strip Unused Function Prototypes</td>
<td><code>(-strip-dead-prototypes)</code></td>
</tr>
<tr>
<td>Strip all llvm.dbg.declare intrinsics</td>
<td><code>(-strip-debug-declare)</code></td>
</tr>
<tr>
<td>Strip all symbols, except dbg symbols, from a module</td>
<td><code>(-strip-nondebug)</code></td>
</tr>
<tr>
<td>Tail Call Elimination</td>
<td><code>(-tailcallelim)</code></td>
</tr>
</tbody>
</table>
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcscsa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
## High-level optimizations (clang)

### Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipscscp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcsssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccpp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcsssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifcfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccp)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)

- Delete dead loops (-loop-deletion)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote 'by reference' arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipscscp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote 'by reference' arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipsccpp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcsssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- 
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplificfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
High-level optimizations (clang)

Transform passes

- Aggressive Dead Code Elimination (-adce)
- Inliner for always_inline functions (-always-inline)
- Promote ‘by reference’ arguments to scalars (-argpromotion)
- Basic-Block Vectorization (-bb-vectorize)
- Profile Guided Basic Block Placement (-block-placement)
- Break critical edges in CFG (-break-crit-edges)
- Optimize for code generation (-codegenprepare)
- Merge Duplicate Global Constants (-constmerge)
- Simple constant propagation (-constprop)
- Dead Code Elimination (-dce)
- Dead Argument Elimination (-deadargelim)
- Dead Type Elimination (-deadtypeelim)
- Dead Instruction Elimination (-die)
- Dead Store Elimination (-dse)
- Deduce function attributes (-functionattrs)
- Dead Global Elimination (-globaldce)
- Global Variable Optimizer (-globalopt)
- Global Value Numbering (-gvn)
- Canonicalize Induction Variables (-indvars)
- Function Integration/Inlining (-inline)
- Combine redundant instructions (-instcombine)
- Internalize Global Symbols (-internalize)
- Interprocedural constant propagation (-ipconstprop)
- Interprocedural Sparse Conditional Constant Propagation (-ipscscp)
- Jump Threading (-jump-threading)
- Loop-Closed SSA Form Pass (-lcssa)
- Loop Invariant Code Motion (-licm)
- Delete dead loops (-loop-deletion)
- Extract loops into new functions (-loop-extract)
- Extract at most one loop into a new function (-loop-extract-single)
- Loop Strength Reduction (-loop-reduce)
- Rotate Loops (-loop-rotate)
- Canonicalize natural loops (-loop-simplify)
- Unroll loops (-loop-unroll)
- Unswitch loops (-loop-unswitch)
- Lower atomic intrinsics to non-atomic form (-loweratomic)
- Lower invokes to calls, for unwindless code generators (-lowerinvoke)
- Lower SwitchInsts to branches (-lowerswitch)
- Promote Memory to Register (-mem2reg)
- MemCpy Optimization (-memcpyopt)
- Merge Functions (-mergefunc)
- Unify function exit nodes (-mergereturn)
- Partial Inliner (-partial-inliner)
- Remove unused exception handling info (-prune-eh)
- Reassociate expressions (-reassociate)
- Demote all values to stack slots (-reg2mem)
- Scalar Replacement of Aggregates (-sroa)
- Sparse Conditional Constant Propagation (-sccp)
- Simplify the CFG (-simplifycfg)
- Code sinking (-sink)
- Strip all symbols from a module (-strip)
- Strip debug info for unused symbols (-strip-dead-debug-info)
- Strip Unused Function Prototypes (-strip-dead-prototypes)
- Strip all llvm.dbg.declare intrinsics (-strip-debug-declare)
- Strip all symbols, except dbg symbols, from a module (-strip-nondebug)
- Tail Call Elimination (-tailcallelim)
Common optimization passes together (-O2)

example.c:

```c
unsigned square(unsigned x)
{
    unsigned sum = 0, tmp;
    for (unsigned i = 1; i < x; i++) {
        tmp = x;
        sum += x;
    }
    return sum + tmp;
}
```

$ opt -S example.ll
```
declare i32 @square(i32) local_unnamed_addr #0 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    store i32 0, i32* %3, align 4
    store i32 1, i32* %5, align 4
    br label %6

    ; <label>:6:
    %7 = load i32, i32* %5, align 4
    %8 = load i32, i32* %2, align 4
    %9 = icmp ult i32 %7, %8
    br i1 %9, label %10, label %18

    ; <label>:10:
    %11 = load i32, i32* %2, align 4
    store i32 %11, i32* %4, align 4
    %12 = load i32, i32* %2, align 4
    %13 = load i32, i32* %3, align 4
    %14 = add i32 %13, %12
    store i32 %14, i32* %5, align 4
    br label %15

    ; <label>:15:
    %16 = load i32, i32* %5, align 4
    %17 = add i32 %16, 1
    store i32 %17, i32* %5, align 4
    br label %6

    ; <label>:18:
    %19 = load i32, i32* %3, align 4
    %20 = load i32, i32* %4, align 4
    %21 = add i32 %19, %20
    ret i32 %21
}
```

$ opt -S -O2 example.ll
```
declare i32 @square(i32) local_unnamed_addr #0 {
    %2 = icmp ugt i32 %0, 1
    %umax = select i1 %2, i32 %0, i32 1
    %3 = mul i32 %umax, %0
    ret i32 %3
}
```
Dead store elimination

**example.c:**

```c
int fun()
{
    int a = 1;
    a = 2;
    return a;
}
```

```bash
$ opt -S example.ll
```

```assembly
define i32 @fun() #0 {
    %1 = alloca i32, align 4
    store i32 1, i32* %1, align 4
    store i32 2, i32* %1, align 4
    %2 = load i32, i32* %1, align 4
    ret i32 %2
}
```

```bash
$ opt -S -dse example.ll
```

```assembly
define i32 @fun() #0 {
    %1 = alloca i32, align 4
    store i32 2, i32* %1, align 4
    %2 = load i32, i32* %1, align 4
    ret i32 %2
}
```
Outline

1 Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2 Linker

3 Execution
Low-level optimizations
Related to a particular hardware

- Instruction Selection
  - Expand ISel Pseudo-instructions
  - Tail Duplication
  - Optimize machine instruction PHIs
  - Merge disjoint stack slots
  - Local Stack Slot Allocation
  - Remove dead machine instructions
  - Early If-Conversion
  - Machine InstCombiner
  - Machine Loop Invariant Code Motion
  - Machine Common Subexpression Elimination
  - Machine code sinking
  - Peephole Optimizations
  - Remove dead machine instructions
  - X86 LEA Optimize
  - X86 Optimize Call Frame
  - Process Implicit Definitions
  - Live Variable Analysis
  - Machine Natural Loop Construction
  - Eliminate PHI nodes for register allocation
  - Two-Address instruction pass
  - Simple Register Coalescing
  - Machine Instruction Scheduler

- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations
Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler
- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations
Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler

- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations

Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- **Peephole Optimizations**
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler

- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations

Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler

- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations
Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler
- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations

Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler
- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations
Related to a particular hardware

- Instruction Selection
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Remove dead machine instructions
- X86 LEA Optimize
- X86 Optimize Call Frame
- Process Implicit Definitions
- Live Variable Analysis
- Machine Natural Loop Construction
- Eliminate PHI nodes for register allocation
- Two-Address instruction pass
- Simple Register Coalescing
- Machine Instruction Scheduler

- Greedy Register Allocator
- Virtual Register Rewriter
- Stack Slot Coloring
- Machine Loop Invariant Code Motion
- X86 FP Stackifier
- Shrink Wrapping analysis
- Prologue/Epilogue Insertion & Frame Finalization
- Control Flow Optimizer
- Tail Duplication
- Machine Copy Propagation Pass
- Post-RA pseudo instruction expansion pass
- X86 pseudo instruction expansion pass
- Post RA top-down list latency scheduler
- Analyze Machine Code For Garbage Collection
- Branch Probability Basic Block Placement
- Execution dependency fix
- X86 vzeroupper inserter
- X86 Atom pad short functions
- X86 LEA Fixup
- Contiguously Lay Out Funclets
- StackMap Liveness Analysis
- Live DEBUG_VALUE analysis
Low-level optimizations

After Instruction Selection:

Frame Objects:
   fi#0: size=4, align=4, at location [SP+8]

Function Live Ins: %EDI in %vreg0

BB#0: derived from LLVM BB %1
   Live Ins: %EDI
   %vreg0<def> = COPY %EDI; GR32:%vreg0
   MOV32mr <fi#0>, 1, %noreg, 0, %noreg, %vreg0; mem:ST4[%2] GR32:%vreg0
   %vreg1<def,tied1> = IMUL32rr %vreg0<tied0>, %vreg0, %EFLAGS<imp-def,dead>; GR32:%vreg1
   %EAX<def> = COPY %vreg1; GR32:%vreg1
   RET 0, %EAX
After Live Variable Analysis:

Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]

Function Live

Ins: %EDI in %vreg0

BB#0: derived from LLVM BB %1

Live

Ins: %EDI

%vreg0<def> = COPY %EDI<kill>; GR32:%vreg0
MOV32mr <fi#0>, 1, %noreg, 0, %noreg, %vreg0; mem:ST4[%2] GR32:%vreg0
%vreg1<def,tied1> = IMUL32rr %vreg0<kill,tied0>, %vreg0, %EFLAGS<imp-def,dead>;
%EAX<def> = COPY %vreg1<kill>; GR32:%vreg1
RET 0, %EAX<kill>
After Two-Address instruction pass:

Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]

Function Live Ins: %EDI in %vreg0

BB#0: derived from LLVM BB %1
  Live Ins: %EDI
  %vreg0<def> = COPY %EDI<kill>; GR32:%vreg0
  MOV32mr <fi#0>, 1, %noreg, 0, %noreg, %vreg0; mem:ST4[%2] GR32:%vreg0
  %vreg1<def> = COPY %vreg0<kill>; GR32:%vreg1,%vreg0
  %vreg1<def,tied1> = IMUL32rr %vreg1<tied0>, %vreg1, %EFLAGS<imp-def,dead>; GR32:%vreg1
  %EAX<def> = COPY %vreg1<kill>; GR32:%vreg1
  RET 0, %EAX<kill>
Low-level optimizations

After Simple Register Coalescing:

Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
Function Live Ins: %EDI in %vreg0

BB#0: derived from LLVM BB %1
  Live Ins: %EDI
    %vreg1<def> = COPY %EDI; GR32:%vreg1
    MOV32mr <fi#0>, 1, %noreg, 0, %noreg, %vreg1; mem:ST4[%2] GR32:%vreg1
    %vreg1<def,tied1> = IMUL32rr %vreg1<tied0>, %vreg1, %EFLAGS<imp-def,dead>; GR32:%vreg1
    %EAX<def> = COPY %vreg1; GR32:%vreg1
    RET 0, %EAX<kill>
Low-level optimizations

After Virtual Register Rewriter:

Frame Objects:

   fi#0: size=4, align=4, at location [SP+8]

Function Live Ins: %EDI

BB#0: derived from LLVM BB %1

   Live Ins: %EDI

   MOV32mr <fi#0>, 1, %noreg, 0, %noreg, %EDI; mem:ST4[%2]

   %EDI<def, tied1> = IMUL32rr %EDI<kill, tied0>, %EDI, %EFLAGS<imp-def, dead>

   %EAX<def> = COPY %EDI<kill>

   RET 0, %EAX<kill>
Low-level optimizations

After Prologue/Epilogue Insertion & Frame Finalization:

Frame Objects:
- fi#-1: size=8, align=16, fixed, at location [SP-8]
- fi#0: size=4, align=4, at location [SP-12]

Function Live Ins: %EDI

BB#0: derived from LLVM BB %1

Live Ins: %EDI %RBP
- PUSH64r %RBP<kill>, %RSP<imp-def>, %RSP<imp-use>; flags: FrameSetup
- CFI_INSTRUCTION <call frame instruction>
- CFI_INSTRUCTION <call frame instruction>
- %RBP<def> = MOV64rr %RSP; flags: FrameSetup
- CFI_INSTRUCTION <call frame instruction>
- MOV32mr %RBP, 1, %noreg, -4, %noreg, %EDI; mem:ST4[%2]
- %EDI<def,tied1> = IMUL32rr %EDI<kill,tied0>, %EDI, %EFLAGS<imp-def,dead>
- %EAX<def> = COPY %EDI<kill>
- %RBP<def> = POP64r %RSP<imp-def>, %RSP<imp-use>; flags: FrameDestroy
- RET 0, %EAX<kill>
Low-level optimizations

After Post-RA pseudo instruction expansion pass:

Frame Objects:

fi#-1: size=8, align=16, fixed, at location [SP-8]
fi#0: size=4, align=4, at location [SP-12]

Function Live Ins: %EDI

BB#0: derived from LLVM BB %1

Live Ins: %EDI %RBP

PUSH64r %RBP<kill>, %RSP<imp-def>, %RSP<imp-use>; flags: FrameSetup
CFI_INSTRUCTION <call frame instruction>
CFI_INSTRUCTION <call frame instruction>
%RBP<def> = MOV64rr %RSP; flags: FrameSetup
CFI_INSTRUCTION <call frame instruction>
MOV32mr %RBP, 1, %noreg, -4, %noreg, %EDI; mem:ST4[%2]
%EDI<def,tied1> = IMUL32rr %EDI<kill,tied0>, %EDI, %EFLAGS<imp-def,dead>
%EAX<def> = MOV32rr %EDI<kill>
%RBP<def> = POP64r %RSP<imp-def>, %RSP<imp-use>; flags: FrameDestroy
RET 0, %EAX<kill>
After X86 pseudo instruction expansion pass:

Frame Objects:
   fi\#-1: size=8, align=16, fixed, at location [SP-8]
   fi\#0: size=4, align=4, at location [SP-12]

Function Live Ins: %EDI

BB#0: derived from LLVM BB %1
   Live Ins: %EDI %RBP
      PUSH64r %RBP<kill>, %RSP<imp-def>, %RSP<imp-use>; flags: FrameSetup
      CFI_INSTRUCTION <call frame instruction>
      CFI_INSTRUCTION <call frame instruction>
      %RBP<def> = MOV64rr %RSP; flags: FrameSetup
      CFI_INSTRUCTION <call frame instruction>
      MOV32mr %RBP, 1, %noreg, -4, %noreg, %EDI; mem:ST4[%2]
      %EDI<def,tied1> = IMUL32rr %EDI<kill,tied0>, %EDI, %EFLAGS<imp-def,dead>
      %EAX<def> = MOV32rr %EDI<kill>
      %RBP<def> = POP64r %RSP<imp-def>, %RSP<imp-use>; flags: FrameDestroy
      RETQ %EAX<kill>
1. Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2. Linker

3. Execution
Profile-guided optimization

1. Compile your application with -fprofile-generate
2. Run tests of your application, gather profiling data
3. Recompile with -fprofile-use
volatile int x;

- It tells the compiler not to optimize the access to the variable.
- When the variable appears in the source code, load or store instruction appears in the machine code.
- In C, volatile is much weaker than in Java, where it generates barrier and results in non-cached access.
void vecadd(int *a, int *b, int *c, size_t n)
{
    for (size_t i = 0; i < n; ++i)
    {
        a[i] += c[i];
        b[i] += c[i];
    }
}

- c[i] is read twice – see mov (%rdx,%rax,4),%r8d
- Why?

gcc -g -O2 -march=core2 -o veclib.o veclib.c
objdump -d veclib.o

vecadd:
  730:  test %rcx,%rcx
  733:   je    759 <vecadd+0x29>
  735:   xor %eax,%eax
  737:   nopw 0x0(%rax,%rax,1)
  73e:
  740:  mov (%rdx,%rax,4),%r8d
  744:  add %r8d,(%rdi,%rax,4)
  748:  mov (%rdx,%rax,4),%r8d
  74c:  add %r8d,(%rsi,%rax,4)
  750:  add $0x1,%rax
  754:  cmp %rax,%rcx
  757:  jne 740 <vecadd+0x10>
  759:  retq
void vecadd(int *a, int *b, int *c, size_t n)
{
    for (size_t i = 0; i < n; ++i)
    {
        a[i] += c[i];
        b[i] += c[i];
    }
}

c[i] is read twice – see mov (%rdx,%rax,4),%r8d
Why?

What if c and a point to the same array?
restrict keyword tells the compiler that the pointers of the same type will never point to the same memory location

void vecadd(restrict int *a, restrict int *b,
            restrict int *c, size_t n)

With restrict, the mov instruction at address 748 disappears and the code is about 10% faster.
1 Compiler
   ■ High-level optimizations
   ■ Low-level optimizations
   ■ Miscellaneous

2 Linker

3 Execution
Linker

- Combines multiple modules (object files) together
- Resolves references to symbols from other modules
- Can also perform some optimizations

Basics of working with libraries

```
$ gcc -o file1.o file1.c
$ gcc -o file2.o file2.c
$ ar rvs libmyfiles.a file1.o file2.o  # create static library

$ gcc -o myprog.o myprog.c
$ ld -o myprog myprog.o -lmyfiles

$ gcc -o myprog myprog.c -lmyfiles  # shortcut
```
Resolving references

```c
extern int var; // variable in another .c file
int func();     // function in another .c file
// The above is usually contained in a header file
int foo()
{
  return func() + var;
}
```

- Linker works by reading relocation records stored in the object files
  - Location within the binary section
  - Format (type) of the value
  - Value of what
- Example below:
  - Put at address 0xA in extern.o the address func in PLT32 format.
  - Put at address 0x12 in extern.o the address var in PC32 format (relative to program counter).

```
$ objdump -r extern.o

extern.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET TYPE VALUE
0000000000000000a R_X86_64_PLT32  func-0x0000000000000004
00000000000000012 R_X86_64_PC32    var-0x0000000000000004
```
Linker-related optimizations

- Linker’s work is driven by a “linker script”
  - By modifying the linker script, you can, for example, reorder functions, e.g. put热 functions together to avoid cache self eviction
  - Default linker scripts already contain this:
    ```c
    int hot_function(...) __attribute__((hot));
    ```

- Can perform “Link-time optimization”
  - Unused function removal:
    ```bash
gcc -ffunction-sections ...
ld --gc-sections ...
```
  - Function inlining
  - Interprocedural constant propagation
  - ...
Outline

1. Compiler
   - High-level optimizations
   - Low-level optimizations
   - Miscellaneous

2. Linker

3. Execution
Starting of a binary program (Linux)

1. OS kernel loads binary header(s)

2. For statically linked binaries:
   - sets virtual memory data structures up and jumps to the program entry point

3. For dynamically linked binaries (those who require shared libraries):
   - Reads the name of program interpreter (e.g. /lib64/ld-linux-x86-64.so.2)
   - Loads the interpreter binary
   - Execute the interpreter with binary name as a parameter
     - This allows things like transparently running ARM binaries on x86 via Qemu emulator
Binary interpreter and dynamic linking

- Interpreter’s task is to perform dynamic linking
- Similar to static linking (it uses relocation table), but at runtime
- Linking big libraries with huge amount of symbols (e.g. Qt) is slow
  - Lazy linking
  - Not good for real-time applications
Program execution and memory management

Summary: things are done lazily if possible

- Executed binary is not loaded into memory at the beginning
  - Loading is done lazily as a response to page faults
  - Only those parts of the binary, that are actually “touched” are loaded
  - Other things (e.g. debug information, unused data and code) stay on disk

- Memory allocation is also lazy
  - When an app asks OS for memory, only VM data is set up
  - Only when the memory is touched, it is actually allocated and mapped to the proper place
  - Allows you to allocate more memory that you physically have

- Memory allocations
  - Two levels: OS level and application level
  - Application asks OS for chunks of memory (via brk() or mmap())
  - Application manages this memory as heap (malloc(), new())