

B4M36ESW: Efficient software

Lecture 4: C program compilation and execution Bentley's rules

Michal Sojka

`michal.sojka@cvut.cz`



March 11, 2019

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 Execution

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 Execution

Example

```
// vecadd.c
#define MM 10000000
unsigned a[MM], b[MM], c[MM];

void main()
{
    clock_t start,end;
    for (size_t i = 0; i < MM; ++i)
        a[i] = b[i] = c[i] = i;
    start = clock();
    vecadd(a, b, c, MM);
    end = clock();
    printf("time = %lf\n", (end - start)/
           (double)CLOCKS_PER_SEC);
}

// veclib.c
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];
    }
}
```

```
gcc -Wall -g -O0 -march=core2 -o vecadd *.c
./vecadd
# time = 0.37
```

```
gcc -Wall -g -O2 -march=core2 -o vecadd *.c
./vecadd
# time = 0.12 ~ 300% speedup
```

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

```
vecadd:
    test    %rcx,%rcx
    je     29 <vecadd+0x29> -----|
    xor    %eax,%eax
    nopw   0x0(%rax,%rax,1)         |
                                           |
    mov    (%rdx,%rax,4),%r8d ←-   |
    add   %r8d,(%rdi,%rax,4)      | |
    mov    (%rdx,%rax,4),%r8d     | |
    add   %r8d,(%rsi,%rax,4)      | |
    add   $0x1,%rax                | |
    cmp   %rax,%rcx                | |
    jne   10 <vecadd+0x10> ----'|
    retq  ←-----'
```

Why is the red mov in the code twice?

Pointer aliasing

- Because `c` may **alias** with `a`!
- `vecadd()` must work correctly even when called as `vecadd(a, a, a, MM)`
- Pointer aliasing = multiple pointers of the same type can point to the same memory
 - This prevents certain optimizations
- **restrict** qualifier = promise that pointer parameters of the same type can never alias

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

```
{ ... }
```

```
./vecadd
```

```
# time = 0.10, speedup 10%!
```

- With **restrict**, the second `mov` disappears.

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 Execution

C/C++ compilation

C/C++ compiler typically contains the following parts:

- 1 Compiler **frontend** – converts source code into intermediate representation (IR)
 - Preprocessor
 - Parser
- 2 **Optimization** passes
 - High-level optimizations
 - Low-level optimizations
- 3 A target-dependent **backend**
 - Generates assembly code or machine code
- 4 **Linker** – can be, and usually is, independent of the compiler

Open-source compilers:

- GCC
- LLVM/clang

LLVM has easier to understand code base. GCC improves code readability as well.

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 Execution

Abstract Syntax Tree (AST)

Parser produces AST

example.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:1:1, line:9:1> line:1:10 square 'unsigned int (unsigned int)'
    |^-ParmVarDecl <col:17, col:26> col:26 used x 'unsigned int'
    ^-CompoundStmt <line:2:1, line:9:1>
      |^-DeclStmt <line:3:3, col:24>
        |^-VarDecl <col:3, col:18> col:12 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' Co
        | | |^-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'
```

Intermediate representation (IR)

- AST is converted to IR
- This usually involves “dumb” expansion of templates

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

LLVM intermediate representation
$ clang -S -emit-llvm example.c

define dso_local 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) square(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'
```

Conversion of for loops to IR

■ C code:

```
for (initializer; condition; modifier) {  
    body  
}
```

■ IR “template”:

```
initializer  
goto COND  
COND:  
    if (condition)  
        goto BODY  
    else  
        goto EXIT  
BODY:  
    body  
    modifier  
    goto COND  
EXIT:
```

Intermediate representation vs. assembler

example.c:

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

\$ clang -S -emit-llvm example.c

```
; 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 optnone uwtable
```

```
define dso_local @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 optnone uwtable "correct..."
```

```
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
```

```
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !"clang version 7.0.1-8 (tags/RELEASE_701/final)"}
```

IR is machine independent

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

```
square:
# %bb.0:
    pushq    %rbp
    movq     %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %edi
    imull   -4(%rbp), %edi
    movl    %edi, %eax
    popq    %rbp
    retq

.Lfunc_end0:
    ...
```

\$ llc -O0 -march=arm example.ll

```
square:
@ %bb.0:
    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

.Lfunc_end0:
    ...
```

Assembler generation from IR is detailed later.

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - **Optimization passes**
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 Execution

Optimizations in general

- Many, many options
- `https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Optimize-Options.html`
- `gcc -Q --help=optimizers -O2`
- `https://llvm.org/docs/Passes.html`

High-level optimizations (clang/LLVM)

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/LLVM)

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 (-memcopyopt)**
- Merge Functions (-mergfunc)
- 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:

```
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

```
define dso_local @square(i32) #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* %3, 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

```
define dso_local @square(i32) local_unnamed_addr
    %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 pass

example.c:

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

\$ opt -S example.ll

```
define dso_local 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
}
```

\$ opt -S -dse example.ll

```
define dso_local 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
}
```

Optimization passes – one by one

Source code

example.c:

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

Optimization passes – one by one

Simplify the CFG

```

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) #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:                                ; preds = %10, %1
    %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 %17

; <label>:10:                                ; preds = %6
    %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* %3, align 4
    %15 = load i32, i32* %5, align 4
    %16 = add i32 %15, 1
    store i32 %16, i32* %5, align 4
    br label %6

; <label>:17:                                ; preds = %6
    %18 = load i32, i32* %3, align 4
    %19 = load i32, i32* %4, align 4
    %20 = add i32 %18, %19
    ret i32 %20
}

```

Optimization passes – one by one

SROA

```

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) #0 {
    br label %2

; <label>:2:                                     ; preds = %4, %1
    %.09 = phi i32 [ 0, %1 ], [ %5, %4 ]
    %.0 = phi i32 [ 1, %1 ], [ %6, %4 ]
    %3 = icmp ult i32 %.0, %0
    br i1 %3, label %4, label %7

; <label>:4:                                     ; preds = %2
    %5 = add i32 %.09, %0
    %6 = add i32 %.0, 1
    br label %2

; <label>:7:                                     ; preds = %2
    %8 = add i32 %.09, %0
    ret i32 %8
}

```

Optimization passes – one by one

Global Variable Optimizer

```

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 dso_local @square(i32 @square(i32) local_unnamed_addr #0) {
    br label %2

; <label>:2:                                     ; preds = %4, %1
    %09 = phi i32 [ 0, %1 ], [ %5, %4 ]
    %0 = phi i32 [ 1, %1 ], [ %6, %4 ]
    %3 = icmp ult i32 %0, %0
    br i1 %3, label %4, label %7

; <label>:4:                                     ; preds = %2
    %5 = add i32 %09, %0
    %6 = add i32 %0, 1
    br label %2

; <label>:7:                                     ; preds = %2
    %8 = add i32 %09, %0
    ret i32 %8
}

attributes #0 = { noinline nounwind uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="fal

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{"clang version 7.0.1-8 (tags/RELEASE_701/final)"}

```

Optimization passes – one by one

Simplify the CFG

```

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

; <label>:2:                                     ; preds = %5, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
    %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    br i1 %3, label %5, label %7

; <label>:5:                                     ; preds = %2
    %6 = add i32 %.0, 1
    br label %2

; <label>:7:                                     ; preds = %2
    ret i32 %4
}

```

Optimization passes – one by one

Deduce function attributes

```

; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

; <label>:2:                                     ; preds = %5, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
    %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    br i1 %3, label %5, label %7

; <label>:5:                                     ; preds = %2
    %6 = add i32 %.0, 1
    br label %2

; <label>:7:                                     ; preds = %2
    ret i32 %4
}

```


Optimization passes – one by one

Loop-Closed SSA Form Pass

```

; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

; <label>:2:                                     ; preds = %5, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %5 ]
    %.0 = phi i32 [ 1, %1 ], [ %6, %5 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    br i1 %3, label %5, label %7

; <label>:5:                                     ; preds = %2
    %6 = add i32 %.0, 1
    br label %2

; <label>:7:                                     ; preds = %2
    %.lcssa = phi i32 [ %4, %2 ]
    ret i32 %.lcssa
}

```

Optimization passes – one by one

Rotate Loops

```

; Preheader:
  br label %2

; Loop:
; <label>:2:                                ; preds = %2, %1
  %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
  %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.09, %0
  %5 = add i32 %.0, 1
  br i1 %3, label %2, label %6

; Exit blocks
; <label>:6:                                ; preds = %2
  %.lcssa = phi i32 [ %4, %2 ]
  ret i32 %.lcssa

```

Optimization passes – one by one

Combine redundant instructions

```

; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

; <label>:2:                                     ; preds = %2, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
    %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    %5 = add i32 %.0, 1
    br i1 %3, label %2, label %6

; <label>:6:                                     ; preds = %2
    ret i32 %4
}

```

Optimization passes – one by one

Loop-Closed SSA Form Pass

```

; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    br label %2

; <label>:2:                                     ; preds = %2, %1
    %.09 = phi i32 [ 0, %1 ], [ %4, %2 ]
    %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
    %3 = icmp ult i32 %.0, %0
    %4 = add i32 %.09, %0
    %5 = add i32 %.0, 1
    br i1 %3, label %2, label %6

; <label>:6:                                     ; preds = %2
    %1.cssa = phi i32 [ %4, %2 ]
    ret i32 %1.cssa
}

```

Optimization passes – one by one

Induction Variable Simplification

```

; Preheader:
  %2 = icmp ugt i32 %0, 1
  %umax = select i1 %2, i32 %0, i32 1
  br label %3

; Loop:
; <label>:3:                                     ; preds = %3, %1
  %.0 = phi i32 [ 1, %1 ], [ %5, %3 ]
  %4 = icmp ult i32 %.0, %0
  %5 = add i32 %.0, 1
  br i1 %4, label %3, label %6

; Exit blocks
; <label>:6:                                     ; preds = %3
  %7 = mul i32 %0, %umax
  ret i32 %7

```

Optimization passes – one by one

Global Value Numbering

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local i32 @square(i32) local_unnamed_addr #0 {
    %2 = icmp ugt i32 %0, 1
    %umax = select i1 %2, i32 %0, i32 1
    %3 = mul i32 %0, %umax
    ret i32 %3
}
```

Optimization passes – one by one

Combine redundant instructions

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local 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
}
```

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 optimization passes

Source code

example.c:

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

Low-level optimization passes

After Instruction Selection:

Frame Objects:

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

Function Live Ins: \$edi in %0

bb.0 (%ir-block.1):

liveins: \$edi

%0:gr32 = COPY \$edi

%1:gr32 = COPY killed %0:gr32

MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, %1:gr32 :: (store 4 into %ir.2)

%6:gr32 = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

%5:gr32 = IMUL32rm killed %6:gr32, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def \$eflags :: (load 4 from %ir.2)

\$eax = COPY %5:gr32

RETQ implicit \$eax

Low-level optimization passes

After Live Variable Analysis:

Frame Objects:

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

Function Live Ins: \$edi in %0

bb.0 (%ir-block.1):

liveins: \$edi

%0:gr32 = COPY killed \$edi

%1:gr32 = COPY killed %0:gr32

MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, killed %1:gr32 :: (store 4 into %ir.2)

%6:gr32 = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

%5:gr32 = IMUL32rm killed %6:gr32, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead \$eflags :: (load 4 from

\$eax = COPY killed %5:gr32

RETQ implicit killed \$eax

Low-level optimization passes

After Two-Address instruction pass:

Frame Objects:

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

Function Live Ins: \$edi in %0

bb.0 (%ir-block.1):

liveins: \$edi

%0:gr32 = COPY killed \$edi

%1:gr32 = COPY killed %0:gr32

MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, killed %1:gr32 :: (store 4 into %ir.2)

%6:gr32 = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

%5:gr32 = COPY killed %6:gr32

%5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead \$eflags :: (load 4 from %ir.2)

\$eax = COPY killed %5:gr32

RETQ implicit killed \$eax

Low-level optimization passes

After Simple Register Coalescing:

Frame Objects:

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

Function Live Ins: \$edi in %0

0B bb.0 (%ir-block.1):

liveins: \$edi

16B %1:gr32 = COPY \$edi

48B MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, %1:gr32 :: (store 4 into %ir.2)

64B %5:gr32 = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

96B %5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead \$eflags :: (load 4 f

112B \$eax = COPY %5:gr32

128B RETQ implicit killed \$eax

Low-level optimization passes

After Greedy Register Allocator:

Frame Objects:

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

Function Live Ins: \$edi in %0

0B bb.0 (%ir-block.1):

liveins: \$edi

16B %1:gr32 = COPY \$edi

48B MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, %1:gr32 :: (store 4 into %ir.2)

64B %5:gr32 = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

96B %5:gr32 = IMUL32rm %5:gr32, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead \$eflags :: (load 4 f

112B \$eax = COPY %5:gr32

128B RETQ implicit \$eax

Low-level optimization passes

After Virtual Register Rewriter:

Frame Objects:

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

Function Live Ins: \$edi

0B bb.0 (%ir-block.1):

liveins: \$edi

48B MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, killed renamable \$edi :: (store 4 into %ir.2)

64B renamable \$eax = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

96B renamable \$eax = IMUL32rm killed renamable \$eax, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead

128B RETQ implicit \$eax

Low-level optimization passes

After Stack Slot Coloring:

Frame Objects:

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

Function Live Ins: \$edi

bb.0 (%ir-block.1):

liveins: \$edi

MOV32mr %stack.0, 1, \$noreg, 0, \$noreg, killed renamable \$edi :: (store 4 into %ir.2)

renamable \$eax = MOV32rm %stack.0, 1, \$noreg, 0, \$noreg :: (load 4 from %ir.2)

renamable \$eax = IMUL32rm killed renamable \$eax, %stack.0, 1, \$noreg, 0, \$noreg, implicit-def dead \$eflags ::

RETQ implicit \$eax

Low-level optimization passes

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 (%ir-block.1):

```
liveins: $edi
```

```
frame-setup PUSH64r killed $rbp, implicit-def $rsp, implicit $rsp
```

```
CFI_INSTRUCTION def_cfa_offset 16
```

```
CFI_INSTRUCTION offset $rbp, -16
```

```
$rbp = frame-setup MOV64rr $rsp
```

```
CFI_INSTRUCTION def_cfa_register $rbp
```

```
MOV32mr $rbp, 1, $noreg, -4, $noreg, killed renamable $edi :: (store 4 into %ir.2)
```

```
renamable $eax = MOV32rm $rbp, 1, $noreg, -4, $noreg :: (load 4 from %ir.2)
```

```
renamable $eax = IMUL32rm killed renamable $eax, $rbp, 1, $noreg, -4, $noreg, implicit-def dead $eflags :: (
```

```
$rbp = frame-destroy POP64r implicit-def $rsp, implicit $rsp
```

```
CFI_INSTRUCTION def_cfa $rsp, 8
```

```
RETQ implicit $eax
```

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - **Miscellaneous**
- 3 Linker
- 4 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 keyword in C

```
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.

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 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

```
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 the address of `func` in PLT32 format at address `0xA` in `extern.o`.
 - Put the address `var` in PC32 format (relative to program counter) at address `0x12` in `extern.o`.

```
$ 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 hot functions together to avoid cache self eviction
 - Default linker scripts already contain this:

```
int hot_function(...) __attribute__((hot));
```
- Can perform “Link-time optimization”
 - Unused function removal:

```
gcc -ffunction-sections ...  
ld --gc-sections ...
```
 - Function inlining
 - Interprocedural constant propagation
 - ...

Outline

- 1 Motivating example
- 2 C/C++ compiler
 - Frontend
 - Optimization passes
 - High-level optimizations
 - High-level optimizations – Example
 - Low-level optimizations
 - Low-level optimizations – Example
 - Miscellaneous
- 3 Linker
- 4 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 than 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()`)

References

- John Regehr: How Clang Compiles a Function
<https://blog.regehr.org/archives/1605>
- John Regehr: How LLVM Optimizes a Function
<https://blog.regehr.org/archives/1603>