B4M36ESW: Efficient software

Lecture 2: C/C++ program profiling, compilation and execution

Michal Sojka
michal.sojka@cvut.cz

March 6, 2023

Outline

Outline

Profiling

- Profiling: Identifies where your code is slow
- "Premature optimization is the root of all evil"
  — D. Knuth
- Software is complex!
- We want to optimize the bottlenecks, not all code
- Real world codebases are big: Reading all the code is a waste of time (for optimizing)

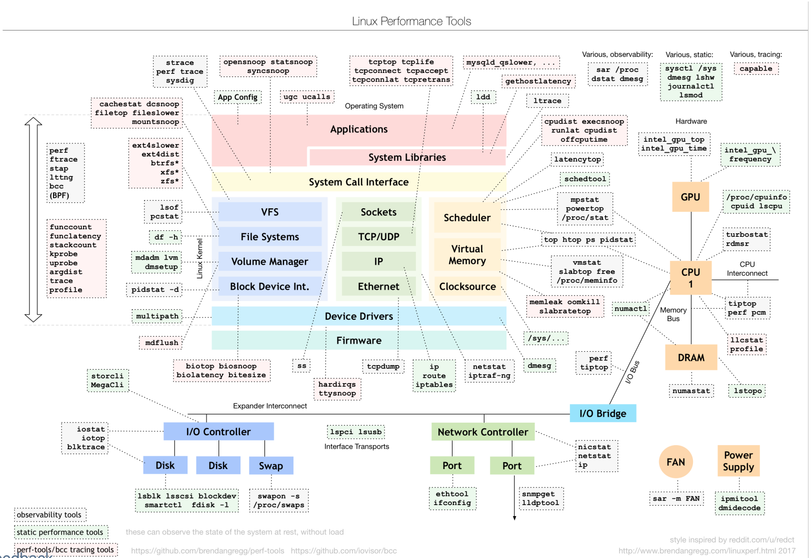
Bottlenecks

There can be many sources of slowness:

- (application) code
- 3rd party libraries
- OS kernel
- memory
- network
- disk
- ...

Finding the source can be difficult...

Linux Performance Tools



Profiling tools

In order to do:	**You can use:**
Manual instrumentation	printf() and similar
Static instrumentation	gprof (GNU profiler)
Dynamic instrumentation	callgrind, cachegrind
Performance counter	oprofile, perf
Heap profiling	massif, google-perftools

Instrumentation = modifying the code to perform measurements

Static instrumentation

gprof usage:

- `gcc -pg ... -o program`
  - Adds profiling code to every function and basic block
- `./program`
  - Runs the program, it generates `gmon.out` file
- `gprof program`

```
Flat profile:

Each sample counts as 0.01 seconds.
%     cumulative self          self    total
time  seconds    seconds calls s/call  s/call  name
33.86 15.52      15.52    1    15.52   15.52   func2
33.82 31.02      15.50    1    15.50   15.50   new_func1
33.29 46.27      15.26    1    15.26   30.75   func1
0.07  46.30      0.03                          main
```

Event sampling

Static instrumentation has problems: overhead, modifies code

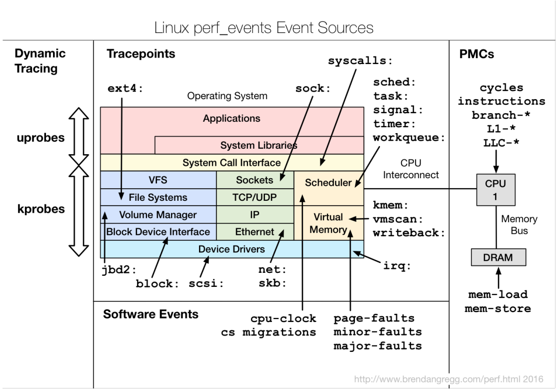
- Basic idea of event sampling
  - When an interesting event occurs, look at where the program executes
  - Result is an histogram of addresses and event counts
- Events
  - Time, cache miss, branch-prediction miss, page fault
- Implementation
  - Timer interrupt $\rightarrow$ upon ISR entry, program address is stored on stack
  - Each event has a counting register in HW
    - Every N (configurable) events, an interrupt is generated

Performance counters

- Hardware inside the CPU (Intel, ARM, ...)
- Software can configure which events to count and when/whether to generate interrupts
- In many cases can be accessed from application code
- Documentation:
  - Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile

Linux tool `perf`

- Can monitor different types of events:
  - HW events (performance counters)
  - SW events (system calls, trace points, ...)
- Can analyze:
  - single application (process + kernel)
  - whole system (all processes + kernel)
- (Re)stores event counts at context switches
- `https://perf.wiki.kernel.org/`



perf usage: stat

- **perf list** – lists available events
- **perf stat -e cycles -e branch-misses -e branches -e cache-misses -e cache-references ./vecadd**

  Collects event counts during execution of the whole program:

```
Performance counter stats for './vecadd':

    1,898,543,656      cycles                                                   (79.98%)
          267,572      branch-misses             #    0.08% of all branches     (79.97%)
      348,090,074      branches                                                 (79.95%)
       20,232,628      cache-misses              #   75.588 % of all cache refs (80.51%)
       26,767,103      cache-references                                         (80.09%)

      0.619472916 seconds time elapsed
```

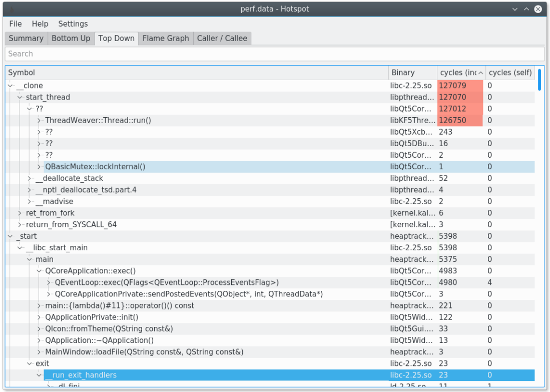
perf usage: record/report

- **perf record -e cycles -e cache-misses ./vecadd**
- **perf record –call-graph ...** – record not only instruction pointer, but also the whole call graph
- **perf report**

```
Samples: 4K of event 'cache-misses', Event count (approx.): 32134369
  Children      Self  Command  Shared Object                Symbol
+   40.71%     0.00%  kcf_vot  [unknown]                    [k] 0x5541d68949564100
+   40.71%     0.00%  kcf_vot  libc-2.29.so                 [.] 0x00007f3ebd54ebbb
+   40.65%     0.00%  kcf_vot  kcf_vot                      [.] main
-   39.58%     0.00%  kcf_vot  kcf_vot                      [.] KCF_Tracker::track
   - KCF_Tracker::track
      - 36.82% ThreadCtx::track
         - 18.89% KCF_Tracker::get_features
            + 9.93% FHoG::extract
              4.55% KCF_Tracker::get_subwindow
              1.79% CNFeat::extract
              1.06% 0x7f3ebd68732f
              0.66% 0x7f3ebd68773d
           9.36% 0x7f3ebd68732f
         + 8.10% KCF_Tracker::GaussianCorrelation::operator()
      + 1.80% KCF_Tracker::train
      + 0.76% cv::Mat::clone
+   36.82%     0.05%  kcf_vot  kcf_vot                      [.] ThreadCtx::track
+   19.94%     0.00%  kcf_vot  kcf_vot                      [.] KCF_Tracker::get_features
+   12.73%     0.00%  kcf_vot  libc-2.29.so                 [.] 0x00007f3ebd68732f
+   12.42%    12.42%  kcf_vot  libc-2.29.so                 [.] 0x000000000015f32f
```

Hotspot – the perf GUI

https://github.com/KDAB/hotspot



Useful resources

- Brendan Gregg's site: `https://www.brendangregg.com/perf.html`
- Denis Bakhvalov's blog: `https://easyperf.net/notes/`
  - and contests: `https://easyperf.net/contest/`
- Performance Matters blog: `https://travisdowns.github.io/`

Outline

Outline

Compiler flags (gcc, clang)

- Documentation is your friend:
  - Command `(p)info gcc`
  - `https://gcc.gnu.org/onlinedocs/`
  - Clang's flags are mostly compatible with gcc
- Generate debugging information: `-g`
- Optimization level: `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (size), `-Og` (debugging)
  - `-O2` is considered "safe", `-O3` may be buggy
  - Individual optimization passes:
    `-ftree-ccp`, `-fast-math`, `-fomit-frame-pointer`, `-ftree-vectorize`, ...
  - Find out which optimizations passes are active for given optimization level: `g++ -Q -O2 --help=optimizers`
- Code generation
  - `-fpic`, `-fpack-struct`, `-fshort-enums`
  - Machine dependent:
    - Generate instructions for given micro-architecture: `-march=haswell`, `-march=skylake` (will not run on older hardware)
    - Use only "older" instructions, but schedule them for for given µarch: `-mtune=haswell`, `-mtune=native`,
    - `-m32`, `-minline-all-stringops`, ...

Outline

Motivating example

```c
// vecadd.c
#define MM 100000000
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    <-------------------------'
```

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

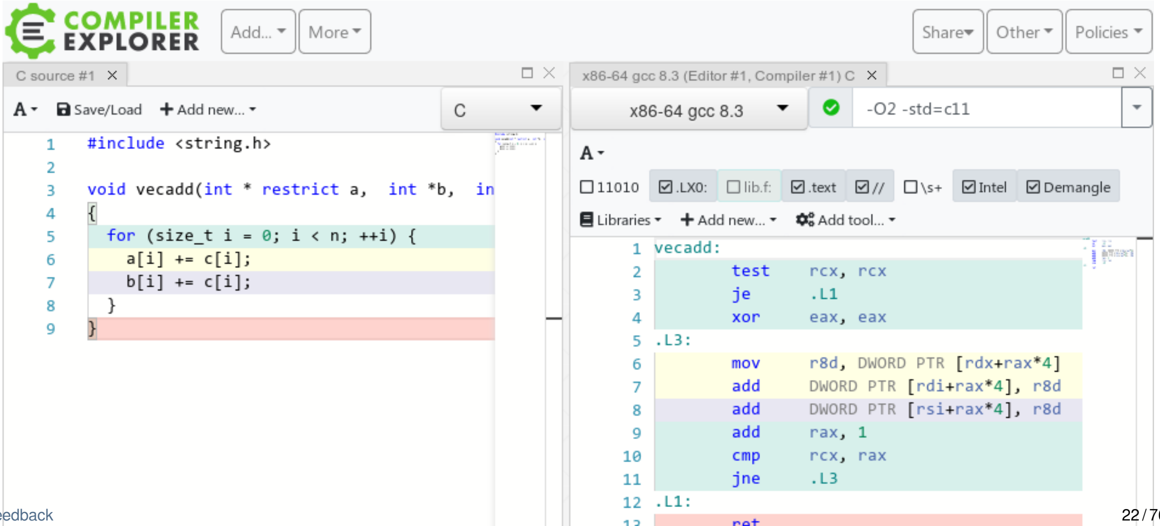
```
void vecadd(int * restrict a, int * b, int * c, size_t n)
{ ... }
```

```
./vecadd
# time = 0.10, speedup 10%!
```

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

Compile Explorer

Play with the example at: `https://godbolt.org/z/opLwvN`



Outline

C/C++ compilation

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

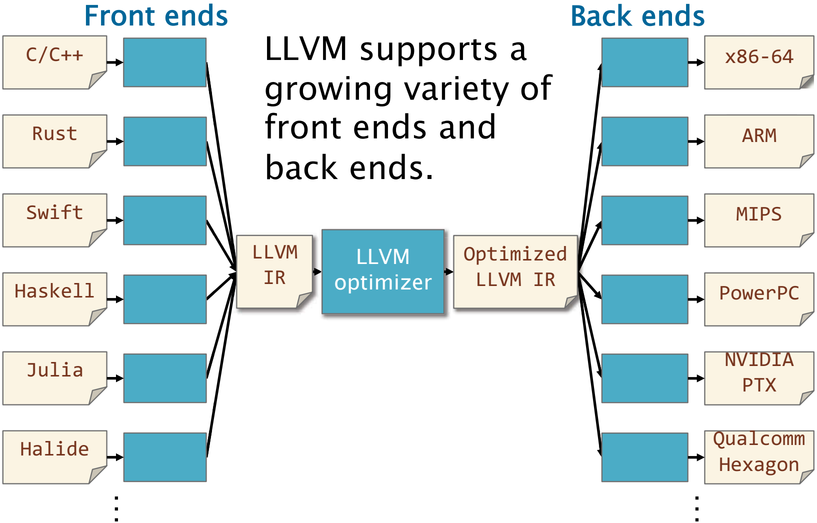
1. Compiler **frontend** – converts source code into intermediate representation (IR)
   - Preprocessor
   - Parser
2. **Semantic checks** – ensuring that the program “makes sense”
   - variables are defined before they’re used,
   - types matches, etc.
   - The compiler constructs a **symbol table**, and adds **attributes** to every expression – these are used in later stages
3. **Optimization** passes
   - High-level optimizations
   - Low-level optimizations
4. A target-dependent **backend**
   - Generates assembly code or machine code
5. **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.

Larger Context of the Compiler



LLVM supports a growing variety of front ends and back ends.

Outline

Abstract Syntax Tree (AST)

Parser produces AST

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:1:1, 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' ComputeResultTy='unsigned int'
    |     |-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 (see below and next slide)

example.c:

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

LLVM intermediate representation
$ clang -S -emit-llvm -O0 example.c

```
define i32 @square(i32 %0) #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
}
```

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

```
TranslationUnitDecl 0xd0ca08 <<invalid sloc>> <invalid sloc>
`-FunctionDecl 0xd4c318 <example.c:1:1, line:4:1> line:1:10 square 'unsigned int (unsigned
  |-ParmVarDecl 0xd4c240 <col:17, col:26> col:26 used x 'unsigned int'
  `-CompoundStmt 0xd4c4a8 <line:2:1, line:4:1>
    `-ReturnStmt 0xd4c498 <line:3:3, col:12>
      `-BinaryOperator 0xd4c478 <col:10, col:12> 'unsigned int' '*'
        |-ImplicitCastExpr 0xd4c448 <col:10> 'unsigned int' <LValueToRValue>
        | `-DeclRefExpr 0xd4c408 <col:10> 'unsigned int' lvalue ParmVar 0xd4c240 'x' 'unsig
        `-ImplicitCastExpr 0xd4c460 <col:12> 'unsigned int' <LValueToRValue>
          `-DeclRefExpr 0xd4c428 <col:12> 'unsigned int' lvalue ParmVar 0xd4c240 'x' 'unsig
```

- **Integer multiplication template:**
  evaluate the first operand and `load` it to a register
  evaluate the second operand and `load` it to a register
  insert `mul` instruction
- **return template:**
  evaluate the operand and `load` it to a register
  insert `ret` instruction

Conversion of for-loops to IR

C code:

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

IR “template”:

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

Intermediate representation vs. assembler

example.c:

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

$ clang -S -emit-llvm -O0 example.c

```llvm
; ModuleID = 'example.c'
source_filename = "example.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-..."
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone sspstrong uwtabl...
define i32 @square(i32 %0) #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 sspstrong uwtabl...

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.1.0"}
```

IR is machine independent

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

```asm
square:                              ...
# %bb.0:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %eax
    imull   -4(%rbp), %eax
    popq    %rbp
    retq
.Lfunc_end0:
                                     ...
```

$ llc -O0 -march=arm example.ll

```asm
square:
@ %bb.0:
    sub     sp, sp, #4
    str     r0, [sp]
    ldr     r0, [sp]
    mul     r1, r0, r0
    mov     r0, r1
    add     sp, sp, #4
    mov     pc, lr
.Lfunc_end0:
                                     ...
```

Outline

Analysis passes – add information for use in other passes (clang/LLVM)

- 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)

Outline

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`

Compiler optimizations ~~New Bentley Rules~~

Data structures

- ~~Packing and encoding~~
- ~~Augmentation~~
- ~~Precomputation~~
- ~~Compile-time initialization~~
- ~~Caching~~
- ~~Lazy evaluation~~
- ~~Sparsity~~

Loops

- Hoisting
- ~~Sentinels~~
- Loop unrolling
- Loop fusion
- Eliminating wasted iterations

Logic

- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Short-circuiting
- Ordering tests
- ~~Creating a fast path~~
- Combining tests

Functions

- Inlining
- Tail-recursion elimination
- ~~Coarsening recursion~~

More Compiler Optimizations

Data structures

- Register allocation
- Memory to registers
- Scalar replacement of aggregates
- Alignment

Loops

- Vectorization
- Unswitching
- Idiom replacement
- Loop fission
- Loop skewing
- Loop tiling
- Loop interchange

Logic

- Elimination of redundant instructions
- Strength reduction
- Dead-code elimination
- Idiom replacement
- Branch reordering
- Global value numbering

Functions

- Unswitching
- Argument elimination

Moving target: Compiler developers implement new optimization over time.

High-level optimizations (clang/LLVM)

Transform passes (`https://llvm.org/docs/Passes.html`)

- 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)

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 -O0 example.ll

```llvm
define i32 @square(i32 %0) #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, !tbaa !3
  %6 = bitcast i32* %3 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %6) #2
  store i32 0, i32* %3, align 4, !tbaa !3
  %7 = bitcast i32* %4 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %7) #2
  %8 = bitcast i32* %5 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %8) #2
  store i32 1, i32* %5, align 4, !tbaa !3
  br label %9

9:
  %10 = load i32, i32* %5, align 4, !tbaa !3
  %11 = load i32, i32* %2, align 4, !tbaa !3
  %12 = icmp ult i32 %10, %11
  br i1 %12, label %15, label %13

13:
  %14 = bitcast i32* %5 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %14) #2
  br label %23

15:
  %16 = load i32, i32* %2, align 4, !tbaa !3
  store i32 %16, i32* %4, align 4, !tbaa !3
  %17 = load i32, i32* %2, align 4, !tbaa !3
  %18 = load i32, i32* %3, align 4, !tbaa !3
  %19 = add i32 %18, %17
  store i32 %19, i32* %3, align 4, !tbaa !3
  br label %20

20:
  %21 = load i32, i32* %5, align 4, !tbaa !3
```

$ opt -S -O2 example.ll

```llvm
define i32 @square(i32 %0) 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 pass

example.c:

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

$ opt -S -O0 example.ll

```llvm
define i32 @fun() #0 {
  %1 = alloca i32, align 4
  %2 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %2) #2
  store i32 1, i32* %1, align 4, !tbaa !3
  store i32 2, i32* %1, align 4, !tbaa !3
  %3 = load i32, i32* %1, align 4, !tbaa !3
  %4 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %4) #2
  ret i32 %3
}
```

$ opt -S -dse example.ll

```llvm
define i32 @fun() #0 {
  %1 = alloca i32, align 4
  %2 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8*
  store i32 2, i32* %1, align 4, !tbaa !3
  %3 = load i32, i32* %1, align 4, !tbaa !3
  %4 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %
  ret i32 %3
}
```

Optimization passes – one by one

Source code

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;
}
```

Optimization passes – one by one

Simplify the CFG

```
; Function Attrs: nounwind sspstrong uwtable
define i32 @square(i32 %0) #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, !tbaa !3
  %6 = bitcast i32* %3 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %6) #2
  store i32 0, i32* %3, align 4, !tbaa !3
  %7 = bitcast i32* %4 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %7) #2
  %8 = bitcast i32* %5 to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* %8) #2
  store i32 1, i32* %5, align 4, !tbaa !3
  br label %9

9:                                                ; preds = %20, %1
  %10 = load i32, i32* %5, align 4, !tbaa !3
  %11 = load i32, i32* %2, align 4, !tbaa !3
  %12 = icmp ult i32 %10, %11
  br i1 %12, label %20, label %13

13:                                               ; preds = %9
  %14 = bitcast i32* %5 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %14) #2
  %15 = load i32, i32* %3, align 4, !tbaa !3
  %16 = load i32, i32* %4, align 4, !tbaa !3
  %17 = add i32 %15, %16
  %18 = bitcast i32* %4 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %18) #2
  %19 = bitcast i32* %3 to i8*
  call void @llvm.lifetime.end.p0i8(i64 4, i8* %19) #2
  ret i32 %17

20:                                               ; preds = %9
  %21 = load i32, i32* %2, align 4, !tbaa !3
  store i32 %21, i32* %4, align 4, !tbaa !3
  %22 = load i32, i32* %2, align 4, !tbaa !3
  %23 = load i32, i32* %3, align 4, !tbaa !3
  %24 = add i32 %23, %22
  store i32 %24, i32* %3, align 4, !tbaa !3
  %25 = load i32, i32* %5, align 4, !tbaa !3
  %26 = add i32 %25, 1
  store i32 %26, i32* %5, align 4, !tbaa !3
  br label %9
}
```

Optimization passes – one by one

SROA

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

2:                                                ; preds = %6, %1
  %.014 = phi i32 [ 0, %1 ], [ %7, %6 ]
  %.0 = phi i32 [ 1, %1 ], [ %8, %6 ]
  %3 = icmp ult i32 %.0, %0
  br i1 %3, label %6, label %4

4:                                                ; preds = %2
  %5 = add i32 %.014, %0
  ret i32 %5

6:                                                ; preds = %2
  %7 = add i32 %.014, %0
  %8 = add i32 %.0, 1
  br label %2
}
```

Optimization passes – one by one

Global Variable Optimizer

```
; ModuleID = 'example.ll'
  source_filename = "example.c"
  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
  target triple = "x86_64-unknown-linux-gnu"

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

  2:                                                ; preds = %6, %1
    %.014 = phi i32 [ 0, %1 ], [ %7, %6 ]
    %.0 = phi i32 [ 1, %1 ], [ %8, %6 ]
    %3 = icmp ult i32 %.0, %0
    br i1 %3, label %6, label %4

  4:                                                ; preds = %2
    %5 = add i32 %.014, %0
    ret i32 %5

  6:                                                ; preds = %2
    %7 = add i32 %.014, %0
    %8 = add i32 %.0, 1
    br label %2
  }

  attributes #0 = { nounwind sspstrong uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="none" "less-precise-fpmad"="fals

  !llvm.module.flags = !{!0, !1}
  !llvm.ident = !{!2}

  !0 = !{i32 1, !"wchar_size", i32 4}
  !1 = !{i32 7, !"PIC Level", i32 2}
  !2 = !{!"clang version 11.1.0"}
```

Optimization passes – one by one

Simplify the CFG

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

2:                                                ; preds = %6, %1
  %.014 = phi i32 [ 0, %1 ], [ %4, %6 ]
  %.0 = phi i32 [ 1, %1 ], [ %7, %6 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.014, %0
  br i1 %3, label %6, label %5

5:                                                ; preds = %2
  ret i32 %4

6:                                                ; preds = %2
  %7 = add i32 %.0, 1
  br label %2
}
```

Optimization passes – one by one

Value Propagation

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

2:                                                ; preds = %6, %1
  %.014 = phi i32 [ 0, %1 ], [ %4, %6 ]
  %.0 = phi i32 [ 1, %1 ], [ %7, %6 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.014, %0
  br i1 %3, label %6, label %5

5:                                                ; preds = %2
  ret i32 %4

6:                                                ; preds = %2
  %7 = add nuw i32 %.0, 1
  br label %2
}
```

Optimization passes – one by one

Loop-Closed SSA Form Pass

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

2:                                                ; preds = %6, %1
  %.014 = phi i32 [ 0, %1 ], [ %4, %6 ]
  %.0 = phi i32 [ 1, %1 ], [ %7, %6 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.014, %0
  br i1 %3, label %6, label %5

5:                                                ; preds = %2
  %.lcssa = phi i32 [ %4, %2 ]
  ret i32 %.lcssa

6:                                                ; preds = %2
  %7 = add nuw i32 %.0, 1
  br label %2
}
```

Optimization passes – one by one

Rotate Loops

```
; Preheader:
    br label %2

  ; Loop:
  <badref>:                                               ; preds = %2, %1
    %.014 = phi i32 [ 0, %1 ], [ %4, %2 ]
    %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
    <badref> = icmp ult i32 %.0, %0
    <badref> = add i32 %.014, %0
    <badref> = add nuw i32 %.0, 1
    br i1 %3, label %2, label %6

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

Optimization passes – one by one

Combine redundant instructions

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

2:                                                ; preds = %2, %1
  %.014 = phi i32 [ 0, %1 ], [ %4, %2 ]
  %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.014, %0
  %5 = add nuw i32 %.0, 1
  br i1 %3, label %2, label %6

6:                                                ; preds = %2
  ret i32 %4
}
```

Optimization passes – one by one

Loop-Closed SSA Form Pass

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

2:                                                ; preds = %2, %1
  %.014 = phi i32 [ 0, %1 ], [ %4, %2 ]
  %.0 = phi i32 [ 1, %1 ], [ %5, %2 ]
  %3 = icmp ult i32 %.0, %0
  %4 = add i32 %.014, %0
  %5 = add nuw i32 %.0, 1
  br i1 %3, label %2, label %6

6:                                                ; preds = %2
  %.lcssa = phi i32 [ %4, %2 ]
  ret i32 %.lcssa
}
```

Optimization passes – one by one

Induction Variable Simplification

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

  ; Loop:
  <badref>:                                          ; preds = %3, %1
    br i1 false, label %3, label %4

  ; Exit blocks
  <badref>:                                          ; preds = %3
    <badref> = mul i32 %0, %umax
    ret i32 %5
```

Optimization passes – one by one

Global Value Numbering

```
; Function Attrs: norecurse nounwind readnone sspstrong uwtable
define i32 @square(i32 %0) 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: norecurse nounwind readnone sspstrong uwtable
  define i32 @square(i32 %0) 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:

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

Low-level optimization passes

After Instruction Selection:

```
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY $edi
  %1:gr32 = IMUL32rr %0:gr32(tied-def 0), %0:gr32, implicit-def dead $eflags
  $eax = COPY %1:gr32
  RET 0, $eax
```

Low-level optimization passes

After Live Variable Analysis:

```
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY killed $edi
  %1:gr32 = IMUL32rr killed %0:gr32(tied-def 0), %0:gr32, implicit-def dead $eflags
  $eax = COPY killed %1:gr32
  RET 0, killed $eax
```

Low-level optimization passes

After Two-Address instruction pass:

```
Function Live Ins: $edi in %0

bb.0 (%ir-block.1):
  liveins: $edi
  %0:gr32 = COPY killed $edi
  %1:gr32 = COPY killed %0:gr32
  %1:gr32 = IMUL32rr %1:gr32(tied-def 0), %1:gr32, implicit-def dead $eflags
  $eax = COPY killed %1:gr32
  RET 0, killed $eax
```

Low-level optimization passes

After Simple Register Coalescing:

```
Function Live Ins: $edi in %0

0B          bb.0 (%ir-block.1):
            liveins: $edi
16B            %1:gr32 = COPY $edi
48B            %1:gr32 = IMUL32rr %1:gr32(tied-def 0), %1:gr32, implicit-def dead $eflags
64B            $eax = COPY %1:gr32
80B            RET 0, killed $eax
```

Low-level optimization passes

After Greedy Register Allocator:

```
Function Live Ins: $edi in %0

0B          bb.0 (%ir-block.1):
            liveins: $edi
16B           %1:gr32 = COPY $edi
48B           %1:gr32 = IMUL32rr %1:gr32(tied-def 0), %1:gr32, implicit-def dead $eflags
64B           $eax = COPY %1:gr32
80B           RET 0, $eax
```

Low-level optimization passes

After Virtual Register Rewriter:

```
Function Live Ins: $edi

0B          bb.0 (%ir-block.1):
            liveins: $edi
16B           renamable $eax = COPY $edi
48B           renamable $eax = IMUL32rr killed renamable $eax(tied-def 0), renamable $eax, implicit-def dead $eflags
80B           RET 0, $eax
```

Low-level optimization passes

After Machine Copy Propagation Pass:

```
Function Live Ins: $edi

bb.0 (%ir-block.1):
  liveins: $edi
  renamable $eax = COPY $edi
  renamable $eax = IMUL32rr killed renamable $eax(tied-def 0), $edi, implicit-def dead $eflags
  RET 0, $eax
```

Low-level optimization passes

After Post-RA pseudo instruction expansion pass:

```
Function Live Ins: $edi

bb.0 (%ir-block.1):
  liveins: $edi
  $eax = MOV32rr $edi
  renamable $eax = IMUL32rr killed renamable $eax(tied-def 0), $edi, implicit-def dead $eflags
  RET 0, $eax
```

Low-level optimization passes

After X86 pseudo instruction expansion pass:

```
Function Live Ins: $edi

bb.0 (%ir-block.1):
  liveins: $edi
  $eax = MOV32rr $edi
  renamable $eax = IMUL32rr killed renamable $eax(tied-def 0), $edi, implicit-def dead $eflags
  RETQ $eax
```

Outline

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/C++, volatile is much weaker than in Java, where it generates a barrier and results in a non-cached access.

Outline

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. Each record contains:
  - Value 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
000000000000000a R_X86_64_PLT32    func-0x0000000000000004
0000000000000012 R_X86_64_PC32     var-0x0000000000000004
```

Linker-related optimizations

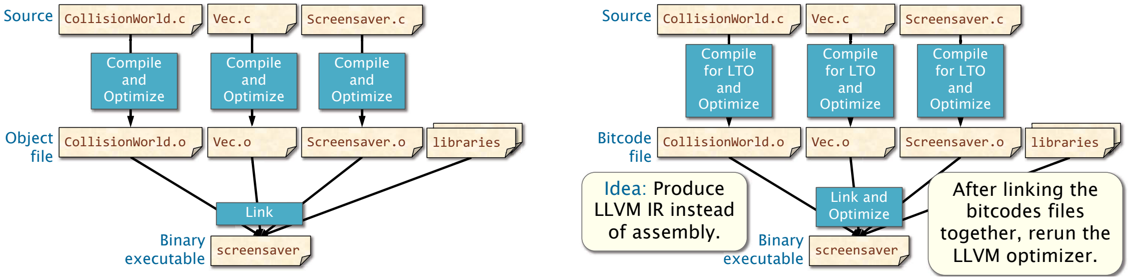
- Linker's work is driven by a "linker script"
  - By modifying the linker script, you can, for example, reorder functions, e.g. put hot functions together to avoid cache self eviction
  - Default linker scripts already support this:

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

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

Link-Time Optimization (LTO)

- Traditionally, compilers optimizes code only within a single file (or compilation unit).
- Modern compilers support link-time optimization, where certain optimization passes can be performed across compilation unit boundaries.



Outline

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
  - Solution: Lazy linking
    - linking (finding a function address and updating the `call` instruction) is done at time of first call
    - Not good for real-time applications
    - Lazy linking can be disabled by setting `LD_BIND_NOW` environment variable (see `man ld-linux.so` on GNU/Linux system)

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 virtual memory (VM) data structures in the OS kernel is set up
  - Only when the memory is touched (page fault), 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())
    - Programs can use different memory allocators. The default one may not be the fastest one for your application.

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
- MIT 6.172 Compiler Lecture