HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 19: Optimization

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

Announcements

- HW5: Oat v.2 out
  - Due in 2 weeks
- HW6 will be released next week
  - Implementing optimizations! (and more)

Today

- Optimizations
  - Safety
  - Constant folding
  - Algebraic simplification
    - Strength reduction
  - Constant propagation
  - Copy propagation
  - Dead code elimination
  - Inlining and specialization
    - Recursive function inlining
  - Tail call elimination
  - Common subexpression elimination

Optimizations

- The code generated by our OAT compiler so far is pretty inefficient.
  - Lots of redundant moves.
  - Lots of unnecessary arithmetic instructions.
- Consider this OAT program:

```
int foo(int w) {
  var x = 3 + 5;
  var y = x * w;
  var z = y - 0;
  return z * 4;
}
```

Unoptimized vs. Optimized Output

```
.globl _foo
_foo:
    pushl %ebp
    movl %esp, %ebp
    subl $64, %esp
__fresh2:
    leal -64(%ebp), %eax
    movl %eax, -48(%ebp)
    movl 8(%ebp), %eax
    movl %eax, %ecx
    movl -48(%ebp), %eax
    movl %ecx, (%eax)
    movl $3, %eax
    movl %eax, -44(%ebp)
    movl $5, %eax
    movl %eax, %ecx
    addl %ecx, -44(%ebp)
    leal -60(%ebp), %eax
    movl %eax, -40(%ebp)
    movl -44(%ebp), %eax
    movl %eax, %ecx
```

- Hand optimized code:

```
_foo:
    shlq    $5, %rdi
    movq    %rdi, %rax
    ret
```

- Function `foo` may be inlined by the compiler, so it can be implemented by just **one instruction**!

Why do we need optimizations?

- To help programmers…
  - They write modular, clean, high-level programs
  - Compiler generates efficient, high-performance assembly
- Programmers don't write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. `A[i][j] = A[i][j] + 1`
- Architectural independence
  - Optimal code depends on features not expressed to the programmer
  - Modern architectures assume optimization
- Different kinds of optimizations:
  - Time: improve execution speed
  - Space: reduce amount of memory needed
  - Power: lower power consumption (e.g. to extend battery life)

Some caveats

- Optimization are code transformations:
  - They can be applied at any stage of the compiler
  - They must be safe – they shouldn't change the meaning of the program.
- In general, optimizations require some program analysis:
  - To determine if the transformation really is safe
  - To determine whether the transformation is cost effective
- "Optimization" is misnomer
  - Typically no guarantee transformations will improve performance, nor that compilation will produce optimal code
- This course: most common and valuable performance optimizations
  - See Muchnick "Advanced Compiler Design and Implementation" for ~10 chapters about optimization

Constant Folding

- Idea: If operands are known at compile type, perform the operation statically.
- `int x = (2+3) * y` ➔ `int x = 5 * y`
- `b & false` ➔ `false`

Constant Folding

```
int x = (2+3) * y  ➔  int x = 5 * y
```

- What performance metric does it intend to improve?
  - In general, the question of whether an optimization improves performance is undecidable.
- At which compilation step can it be applied?
  - Intermediate Representation
  - Can be performed after other optimizations that create constant expressions.

Constant Folding

`int x = (2+3) * y` → `int x = 5 * y`

- When is it safely applicable?
  - For Boolean values, yes.
  - For integers, almost always yes.
    - An exception: division by zero.
  - For floating points, use caution.
    - Example: rounding
- General notes about safety:
  - Whether an optimization is safe depends on language semantics.
    - Languages that provide weaker guarantees to the programmer permit more optimizations, but have more ambiguity in their behavior.
  - Is there a formal proof for safety?

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules.
- Identities:
  - `a * 1 → a` `a * 0 → 0`
  - `a + 0 → a` `a – 0 → a`
  - `b | false → b` `b & true → b`
- Reassociation & commutativity:
  - `(a + b) + c → a + (b + c)`
  - `a + b → b + a`

Algebraic Simplification

- Combined with Constant Folding:
  - `(a + 1) + 2` ➔ `a + (1 + 2)` ➔ `a + 3`
  - `(2 + a) + 4` ➔ `(a + 2) + 4` ➔ `a + (2 + 4)` ➔ `a + 6`
- Iteration of these optimizations is useful…
  - How much?

Strength Reduction

- Replace expensive op with cheaper op:
  - `a * 4` → `a << 2`
  - `a * 7` → `(a << 3) – a`
  - `a / 32767` → `(a >> 15) + (a >> 30)`
- So, the effectiveness of this optimization depends on the architecture.

Constant Propagation

- If the value of a variable is known to be a constant, replace the use of the variable by that constant.
- Value of the variable must be propagated forward from the point of assignment.
  - This is a substitution operation.
- Example:

```
int x = 5;
int y = x * 2;
int z = a[y];
```

➔

```
int y = 5 * 2;
int z = a[y];
```

➔

```
int y = 10;
int z = a[y];
```

➔

```
int z = a[10];
```

- To be most effective, constant propagation can be interleaved with constant folding.

Constant Propagation

- For safety, it requires a data-flow analysis.
  - Next lecture!
- What performance metric does it intend to improve?
- At which compilation step can it be applied?
- What is the computational complexity of this optimization?

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;
if (x > 1) {
  x = x * f(x – 1);
}
```

➔

```
x = y;
if (y > 1) {
  x = y * f(y – 1);
}
```

- Can make the first assignment to x **dead code** (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y // x is dead!
...          // x never used
x = z * z
```

➔

```
...
x = z * z
```

- A variable is **dead** if it is never used after it is defined.
  - Computing such **definition** and **use** information is an important component of compiler
- Dead variables can be created by other optimizations…
- Code for computing the value of a dead variable can be dropped.

Dead Code Elimination

- Is it always safely applicable?
  - Only if that code is **pure** (i.e. it has no externally visible side effects).
    - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket, ...
    - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Unreachable Code Elimination

- Basic blocks not reachable by any trace leading from the starting basic block are **unreachable** and can be deleted.
- At which compilation step can it be applied?
  - IR or assembly level
- What performance metric does it intend to improve?
  - Improves instruction cache utilization.

Common Subexpression Elimination

- Idea: replace an expression with previously stored evaluations of that expression.
- Example:

```
[a + i*4] = [a + i*4] + 1
```

  - Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- As an example, consider function:

```
void f(int[] a, int[] b, int[] c) {
   int j = …; int i = …; int k = …;
   b[j] = a[i] + 1;
   c[k] = a[i];
   return;
}
```

- The following optimization that shares expression `a[i]` is unsafe… Why?

```
void f(int[] a, int[] b, int[] c) {
   int j = …; int i = …; int k = …;
   t = a[i];
   b[j] = t + 1;
   c[k] = t;
   return;
}
```

Common Subexpression Elimination

- Almost always improves performance.
- But sometimes…
  - It might be less expensive to recompute an expression, rather than to allocate another register to hold its value (or to store it in memory and later reload it).

Loop-invariant Code Motion

- Idea: hoist invariant code out of a loop.

```
while (b) {
  z = y/x;
  …              // y, x not updated
}
```

➔

```
z = y/x;
while (b) {
  …              // y, x not updated
}
```

- What performance metric does it intend to improve?
- Is this always safe?

Optimization Example

```
let a = x ** 2 in
let b = 3 in
let c = x in
let d = c * c in
let e = b * 2 in
let f = a + d in
e * f
```

Copy and constant propagation →

```
let a = x ** 2 in
let d = x * x in
let e = 3 * 2 in
let f = a + d in
e * f
```

Constant folding ↓

```
let a = x ** 2 in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

← *Strength reduction*

```
let a = x * x in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

Common sub-expression elimination ↓

```
let a = x * x in
let d = a in
let e = 6 in
let f = a + d in
e * f
```

Copy and constant propagation →

```
let a = x * x in
let f = a + a in
6 * f
```

Loop Unrolling

- Idea: replace the body of a loop by several copies of the body and adjust the loop-control code.
- Example:
  - Before unrolling:

```
for(int i=0; i<100; i=i+1) {
   s = s + a[i];
}
```

  - After unrolling:

```
for(int i=0; i<99; i=i+2){
   s = s + a[i];
   s = s + a[i+1];
}
```

Loop Unrolling

- What performance metric does it intend to improve?
  - Reduces the overhead of branching and checking the loop-control.
    - But it yields larger loops, which might impact the instruction cache.
- Which loops to unroll and by what factor?
  - Some heuristics:
    - Body with straight-line code.
    - Simple loop-control.
  - Use profiled runs.
- It may improve the effectiveness of other optimizations (e.g., common-subexpression evaluation).

Inlining

- Replace call to a function with function body (rewrite arguments to be local variables).
- Example:

```
int g(int x) { return x + pow(x); }

int pow(int a) {
  int b = 1; int n = 0;
  while (n < a) {b = 2 * b};
  return b;
}
```

→

```
int g(int x) {
  int a = x;
  int b = 1; int n = 0;
  while (n < a) {b = 2 * b};
  tmp = b;
  return x + tmp;
}
```

- Eliminates the stack manipulation, jump, etc.
- May need to rename variable names to avoid **name capture.**
  - Example of what can go wrong?
- Best done at the AST or relatively high-level IR.
  - Enables further optimizations.

Inlining Recursive Functions

- Consider recursive function:

```
f(x,y) = if x < 1 then y
         else x * f(x-1,y)
```

- If we inline it, we essentially just unroll one call:
  - `f(z,8) + 7`
    becomes
    `(if z < 0 then 8 else z*f(z-1,8)) + 7`
  - Can't keep on inlining definition of `f`; will never stop!
- But can still get some benefits of inlining by slight rewriting of recursive function...

Rewriting Recursive Functions for Inlining

- Rewrite function to use a loop pre-header

```
function f(a1,...,an) = e
```

becomes

```
function f(a1,...,an) =
    let function f’(a1,...,an) = e[f↦f']
    in f’(a1,...,an)
```

- Example:

```
function f(x,y) = if x < 1 then y else x * f(x-1,y)

function f(x,y) =
   let function f’(x,y) = if x < 1 then y
                          else x * f’(x-1,y)
   in f’(x,y)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
    let function f’(x,y) = if x < 1 then y
                            else x * f’(x-1,y)
    in f’(x,y)
```

- Remove **loop-invariant arguments**
    - e.g., `y` is invariant in calls to `f’`

```
function f(x,y) =
    let function f’(x) = if x < 1 then y
                          else x * f’(x-1)
    in f’(x)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
   let function f'(x) = if x < 1 then y
                        else x * f'(x-1)
   in f'(x)
```

6+f(4,5) becomes:

```
6 +
(let function f'(x)=
  if x < 1 then 5
   else x * f'(x-1)
in f'(4))
```

Without rewriting `f`,

6+f(4,5) becomes:

```
6 +
(if 4 < 1 then 5
   else 4 *
   f(3,5))
```

Rewriting Recursive Functions for Inlining

- Now inlining recursive function is more useful!
  - Can *specialize* the recursive function!
    - Additional optimizations for the specific arguments can be enabled (e.g., copy propagation, dead code elimination).

When to Inline

- Code inlining might increase the code size.
  - Impact on cache misses.
- Some heuristics for when to inline a function:
  - Expand only function call sites that are called frequently
    - Determine frequency by execution profiler or by approximating statically (e.g., loop depth)
  - Expand only functions with small bodies
    - Copied body won't be much larger than code to invoke function
  - Expand functions that are called only once
    - Dead function elimination will remove the now unused function

Tail Call Elimination

- Consider two recursive functions:

```
let add(m,n) = if (m=0) then n else 1 + add(m-1,n)

let add(m,n) = if (m=0) then n else add(m-1,n+1)
```

- First function: after recursive call to `add`, still have computation to do (i.e., add `1`).
- Second function: after recursive call, nothing to do but return to caller.
  - This is a **tail call**.

Tail Call Elimination

```
let add(m,n) = if (m=0) then n else add(m-1,n+1)
```

Equivalent program in an imperative language

```
int add(int m, int n){
 if (m=0) then
  return n
 else
  return add(m-1,n+1)}
```

Tail Call Elimination

```
int add(int m, int n){
loop:
 if (m=0) then
  return n
 else
  m:=m-1;
  n:=n+1;
  goto loop }
```

Tail Call Elimination

- Steps for applying tail call elimination to a recursive procedure:
  - Replace recursive call by updating the parameters.
  - Branch to the beginning of the procedure.
  - Delete the **`return`**.
- Reuse stack frame!
  - Don't need to allocate new stack frame for recursive call.
- Values of arguments (`n`, `m`) remain in registers.
- Combined with inlining, a recursive function can become as cheap as a while loop.
- Even for non-recursive functions: if last statement is function call (tail call), can still reuse stack frame.

Some Optimizations

Level	Representation	Optimizations
High level	AST	• Inlining
		• Function specialization
	IR	• Constant folding
		• Constant propagation
		• Value numbering
Mid level	Canonical IR	• Dead code elimination
		• Loop-invariant code motion
		• Common sub-expression elimination
		• Strength Reduction
		• Constant folding & propagation
Low level	Abstract assembly	• Branch prediction / optimization
		• Register allocation
		• Loop unrolling
	Assembly	• Cache optimization

Writing Fast Programs In Practice

- Pick the right algorithms and data structures.
  - These have a much bigger impact on performance that compiler optimizations.
  - Reduce # of operations
  - Reduce memory accesses
  - Minimize indirection – it breaks working-set coherence
- **Then** turn on compiler optimizations.
- Profile to determine program hot spots.
- Evaluate whether the algorithm/data structure design works.