Lecture Notes on Basic Optimizations

15-411: Compiler Design
André Platzer

Lecture 14

1 Introduction

Several optimizations are easier to perform on SSA form, because SSA needs less analysis. Some optimizations can even be built into SSA construction. Advanced optimizations need advanced static analysis techniques on IR trees and SSA. In these lecture notes, we only deal with the SSA case formally and leave the extra effort for the non-SSA case informal.

Some of what is covered in these notes is explained in [App98, Chapters 17.2–17.3]. A canonical reference on optimizing compilers is the book by [Muc97], which also contains a brief definition of SSA.

2 Constant Folding

The idea behind constant folding is simple but successful. Whenever there is a subexpression that only involves constant operands then we precompute the result of the arithmetic operation during compilation and just use the result. That is, whenever we find a subexpression $n_1 \odot n_2$ for concrete number constants n_1, n_2 and some operator $\odot$, we compute the numerical result n of the expression $n_1 \odot n_2$ and use n instead. Let us say this subexpression occurs in an assignment $x = n_1 \odot n_2$, which, for clarity, we write $x \leftarrow n_1 \odot n_2$ here.

$$\frac{x \leftarrow n_1 \odot n_2 \quad n_1 \odot n_2 = n}{x \leftarrow n}\ CF$$

For that, of course, we have to be careful with the arithmetical semantic of the target language. That is especially for expressions that raise an overflow

or division by zero error. Large expressions with constant operands seldom occur in source code. But around 85% of constant folding comes from code generated by address arithmetic. Constant folding also becomes possible after other optimizations have been performed like constant propagation.

More advanced constant folding proceeds across basic blocks and takes SSA ϕ functions into account. For instance, if we have an expression in SSA that, after constant propagation, is of the form $n_1 \odot \phi(n_2, n_3)$ for some operator $\odot$ and numerical constants n_1, n_2, n_3 then we can perform constant folding across ϕ:

$$\frac{x \leftarrow n_1 \odot \phi(n_2, n_3) \quad n_{12} = n_1 \odot n_2 \quad n_{13} = n_1 \odot n_3}{x \leftarrow \phi(n_{12}, n_{13})} \; CF_\phi$$

3 Inverse Operations

Another optimization is to delete inverse operations, e.g., for unary minus:

$$\frac{-(-a)}{a} \; Inv$$

Again, inverse operators seldom occur in source code but may still arise after other optimizations have been used.

4 Common Subexpression Elimination (CSE)

The goal of common subexpression elimination (CSE) is to avoid repetitive computation for subexpressions that occur repeatedly. Common subexpressions occur very frequently in address arithmetic or in generated source code. Studies show, for instance, that 60% of all arithmetic is address arithmetic in PL/1 programs. A source code expression $a.x = a.x+1$ for instance yields intermediate code with duplicate address arithmetic

```
// compiled from a.x = a.x + 1
t1 = a + offsetx;
t2 = a + offsetx;
*t2 = *t1 + 1;
```

If two operations always yield the same result then they are semantically equivalent. In SSA, if two expressions e, e' are syntactically identical (i.e., the same operators and the same operands) then they are semantically

equivalent. Beware, however, that this is not the case in other intermediate representations, where static analysis is still necessary to determine if the same operands still hold the same values or may already hold different ones. In SSA we get this information for free from the property that each location is (statically) only assigned once. Thus, wherever it is available, it holds the same value.

Consequently, in SSA form, for syntactically identical subexpressions e, e' we can remove the computation e' and just store and use the result of e if e dominates e' (otherwise the value may not be available). We capture this optimization by the following rule:

$$\frac{\begin{array}{l} l' : t \leftarrow a \odot b \\ l : x \leftarrow a \odot b \\ l' \geq l \end{array}}{\begin{array}{l} l' : t \leftarrow a \odot b \\ l : x \leftarrow t \end{array}} \; CSE$$

In rule (CSE) we replace the operation $x \leftarrow a \odot b$ at location l by $x \leftarrow t$, provided we find the same expression $a \odot b$ at another location l' that dominates l (written $l' \geq l$). We leave the operation l' unchanged. Hence there are multiple premises and multiple conclusions in rule (CSE).

Rule (CSE) is easy to implement on SSA as follows. For each subexpression e of the form $a \odot b$ for some operator $\odot$ at an SSA node k, we need to find all SSA nodes k' that have the subexpression e. The canonical way to solve this is to maintain a hash table for all expressions and lookup each expression e in it. If we are at node k' with expression e and the hash table tells us that expression e occurs at a node k and k dominates k' then we reuse the value of e from k at k'.

```
for each node k of the form a ⊙ b do
  j = look up a ⊙ b in hash table
  if any j dominates k then
      use result from j instead of a ⊙ b in k
  else
      leave k as is and put a ⊙ b into hash table
```

Note that the effect of "use result of j" may depend on the choice of the SSA intermediate representation. For arbitrary SSA representations, the value of an arbitrary subexpression $a \odot b$ may have to be stored into a variable at the dominator j in order to even be accessible at k.

For an SSA representation that only allows one operation for each of the instructions (within a basic block), this is simpler. That is, consider an SSA representation where basic blocks only allows operations of the form $x = a \odot b$ for an operator $\odot$ and variables (but not general expressions) a and b, then only top-level common subexpressions need to be identified and their values will already have been stored in a variable. To illustrate, suppose we are looking at a basic block in which one of the instructions is $y_7 = a \odot b$ for variables a, b then we only need to look for instructions of the form $x = a \odot b$ Thus, if $x_5 = a \odot b$ is one of the instructions in a node j that dominates k, then all we need to do to "use result of j" at k is to replace $y_7 = a \odot b$ in k by $y_7 = x_5$, which, in turn will be eliminated by copy propagation or value numbering.

In order to explain what happens in n-operation SSA, we use the following rule

$$\frac{\begin{array}{l} l' : \Upsilon(a \odot b) \\ l : \Upsilon'(a \odot b) \\ l' \geq l \end{array}}{\begin{array}{l} l' : t \leftarrow a \odot b; \Upsilon(t) \\ l : \Upsilon'(t) \end{array}} \; CSE_n$$

$\Upsilon(t)$ represents an operation context with a subterm argument t (and possibly others). And $\Upsilon(a \odot b)$ represents the same operation context but with $a \odot b$ instead of t. Similarly, $\Upsilon'(t)$ represents another operation context with argument t.

The CSE algorithm can also be integrated into the SSA construction, e.g., SSA by AST Walk, because the construction will yield all dominators of k before k (see SSA lecture). Combining CSE with SSA construction also reduces the storage complexity of the SSA graph. Finally, it helps using normalizing transformations like commutativity and distributivity before CSE.

For non-SSA form, we also have to be extra careful that the variables in the subexpression must always still hold the same value. And we need to store the subexpression in a temporary variable, when the other target may possibly be overwritten.

5 Constant propagation

Constant propagation propagates constant values of expressions like $x_2 = 5$ to all dominated occurrences of x_2. That is we just substitute $x_2 = 5$ into

all dominated occurrences of x_2. We capture this by the following rule (in which n, of course, could be any number literal)

$$\frac{\begin{array}{l} l' : t \leftarrow n \\ l : \Upsilon(t) \\ l' \geq l \end{array}}{\begin{array}{l} l' : t \leftarrow n \\ l : \Upsilon(n) \end{array}} \; ConstProp$$

In non-SSA form, extra care needs to be taken that the value of x_2 cannot be different at the occurrence. Constant propagation is somewhat similar to CSE where the operator $\odot$ is just the constant operator (here 5) that does not take any arguments. It is usually implemented either using the CSE hash tables or implicitly during SSA construction.

For non-SSA we also need to be careful that no other definition of x_2 may possibly reach the statement and that the $x_2 = 5$ definition surely reaches the statement without being overwritten (possibly maybe).

6 Copy propagation

Copy propagation propagates values of copies like $x_2 = y_4$ to all dominated occurrences of x_2. That is we just substitute $x_2 = y_4$ into all dominated occurrences of x_2. We capture this by the copy propagation rule in which y is a simple variable

$$\frac{\begin{array}{l} l' : t \leftarrow y \\ l : \Upsilon(t) \\ l' \geq l \end{array}}{\begin{array}{l} l' : t \leftarrow y \\ l : \Upsilon(y) \end{array}} \; CopyProp$$

In non-SSA form, extra care needs to be taken that the value of x_2 cannot be different at the occurrence. Constant propagation is somewhat similar to CSE where the operator $\odot$ is just the identity operator that only takes one argument. It is usually implemented either using the CSE hash tables or implicitly during SSA construction. The register coalescing optimization during register allocation is a form of copy propagation.

For non-SSA we also need to be careful that no other definition of x_2 may possibly reach the statement and that the $x_2 = y_4$ definition surely reaches the statement without being overwritten (possibly maybe) and that no definition of y_4 may possibly reach the statement.

7 Normalization

Normalizing transformations do not optimize the program itself but help subsequent optimizations find more syntactically identical expressions. They use algebraic laws like associativity and distributivity. Their use is generally restricted by definitions of the evaluation order (for Java), by the exception order (Java, Eiffel), or by the limitations of floating-point arithmetic (which is neither associative nor distributive).

A simple normalization is to use commutativity $a + b = b + a$. But we could use this equation in both directions? Which one do we use? If we use it arbitrarily then we may still miss identical expressions. Instead, we fix an order that will always normalize expressions. We first fix an order $\prec$ on all operators. For instance, the order in which we constructed the various expressions during SSA construction. And then we order commutative operations so that the small operand (with respect to the order $\prec$) comes first:

$$\frac{a + b \quad b \prec a}{b + a}\, C_+ \qquad \frac{a * b \quad b \prec a}{b * a}\, C_*$$

How do we use distributivity $a * (b + c) = (a * b) + (a * c)$? Again we need to decide in which direction we use it. This time we have to fix an order on the operators. Let us fix $* \prec +$ and use

$$\frac{a * b + a * c \quad * \prec +}{a * (b + c)}\, D$$

But here we already have to think carefully about overflows. The operation $b+c$ might overflow the data range even if $a*b+a*c$ does not (e.g., for $a = 0$). On an execution architecture where range overflows trigger exceptions or the program depends on the processor flags being set, using distributivity might be unsafe. When we strictly stick to modular arithmetic and do not compile by relying on flags, however, distributivity is safe.

8 Reaching Expressions

The optimizations above have been presented for SSA intermediate representations, where syntactical identity is the primary criterion for semantical equivalence of terms and where def-use relations are represented explicitly in the SSA representation. So we are done with those optimizations as far as SSA is concerned.

For non-SSA, this is not so easy, because we explicitly have to compute all required information by static analysis. We have already seen how reaching definitions can be computed in a previous lecture on dataflow analysis.

Reaching expressions analysis is very similar to reaching definitions analysis from the dataflow analysis lecture. The difference is essentially that we do not care so much about the variable in which an expression has been stored, but only if the expression could have been computed before already. We say that the expression $a \odot b$ at $l : x \leftarrow a \odot b$ *reaches* a line l' if there is a path of control flow from l to l' during which no part of $a \odot b$ is redefined. In logical language:

- $\mathsf{reaches}(l, a \odot b, l')$ if the expression $a \odot b$ at l reaches l'.

We only need two inference rules to defines this analysis. The first states that an expression reaches any immediate successor. The second expresses that we can propagate a reaching expression to all successors of a line l' we have already reached, unless this line also defines a part of the expression.

$$\frac{\begin{array}{l} l : x \leftarrow a \odot b \\ \mathsf{succ}(l, l') \end{array}}{\mathsf{reaches}(l, a \odot b, l')}\ RE_1 \qquad \frac{\begin{array}{l} \mathsf{reaches}(l, a \odot b, l') \\ \mathsf{succ}(l', l'') \\ \neg\mathsf{def}(l', a) \\ \neg\mathsf{def}(l', b) \end{array}}{\mathsf{reaches}(l, a \odot b, l'')}\ RE_2$$

Reaching expression analysis is only needed for a small subset of all expressions during CSE. Thus, it is usually not performed exhaustively but only selectively as needed for some expressions.

9 Available Expressions

Another analysis that is not obvious except for SSA representations is that of available expressions. Reaching expressions capture the expressions by static analysis that could possibly reach a node. But it is not certain that they will, so we cannot always rely on the expression being available under all circumstances. This is what available expressions analysis captures. Which expressions are available at a point no matter what control path has been taken before.

An expression $a \odot b$ is available at a node k if, on every path from the entry to k, the expression $a \odot b$ is computed at least once and no subexpression of $a \odot b$ has been redefined since the last occurrence of $a \odot b$ on

the path. There is a crucial difference to all the dataflow analyses that we have seen in class before. For available expressions we are not interested in what information *may* be preserved from one location to another because at least one control path provides it (may analysis). We are interested in what information *must* be preserved on all control paths reaching the location so that we can rely on it being present (must analysis).

Unfortunately, must analysis is a tricky match for logic rules, because that keeps adding information, but we cannot (easily) talk about negations. What we would have to say is something like

$$\frac{\neg\exists l'(\mathsf{succ}(l_0, l) \wedge \neg\mathsf{avail}(l_0, a \odot b))}{\mathsf{avail}(l, a \odot b)} \ ?$$

This can still be expressed with logic rules, but it is much more complicated to use them in the right way. We have to saturate appropriately before we interpret negations.

Alternatively, we use a representation of available expression analysis by dataflow equations. We follow the dataflow schema shown in Figure 1 using the definitions from Table 1.

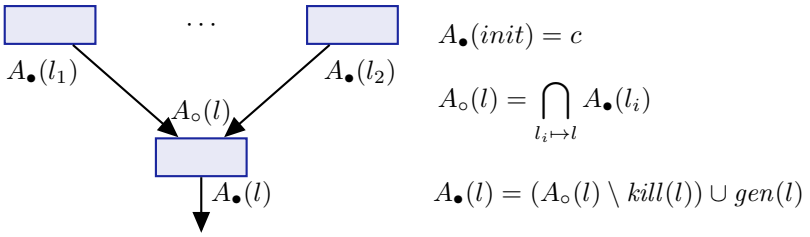

Figure 1: Dataflow analysis schema for available expressions

10 Peephole Optimization

One of the simple-most optimizations is peephole optimization by McKeeman from 1965 [McK65]. Peephole optimization is a postoptimization in the backend after/during instruction selection has been performed. It is an entirely local transformation. The basic idea is to move a sliding peephole, usually of size 2, over the instructions and replace this pair of instructions by cheaper instructions. After precomputing cheaper instruction sequences for the set of all pairs of operations, a simple linear sweep through the instructions is used to optimize each pair. The matter is a little more involved

Table 1: Dataflow analysis definitions for available expressions

Statement l	$gen(l)$	$kill(l)$
$init$	$A_\bullet(init) = \emptyset$	
$x \leftarrow a \odot b$	$\{a \odot b\} \setminus kill(l)$	$\{e \ : \ e$ contains $x\}$
$x \leftarrow *a$	$\{*a\} \setminus kill(l)$	$\{e \ : \ e$ contains $x\}$
$*a \leftarrow b$	$\emptyset$	$\{*z \ : \$ for all $z\}$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\emptyset$	$\emptyset$
l' :	$\emptyset$	$\emptyset$
$x \leftarrow f(p_1, \ldots, p_n)$	$\emptyset$	$\{e \ : \ e$ contains x or any $*z\}$

in the case of conditional jumps, where both possible target locations need to be considered. Peephole optimization can be quite useful for CISC architectures to glue code coming originally from independent AST expressions or to use fancy address modes. For instance, there are address modes that combine

$$a[i]; i{+}{+} \quad \rightsquigarrow \quad a[i{+}{+}]$$

Typical peephole optimizations include

store R, a; load a, R	$\rightsquigarrow$	store R, a	superfluous load
imul 2, R	$\rightsquigarrow$	ashl 2,R	multiplication by constant
iadd x,R; comp 0,R	$\rightsquigarrow$	iadd x, R	superfluous comparisons
if b then x:=y	$\rightsquigarrow$	t:=b; x:=(t) y	IA-64 predicated assignment

The major complication with peephole optimization is its mutual dependency with other instruction selection optimizations like pipeline optimization. Peephole optimization should be done before instruction selection for pipeline optimization but it cannot be done without instructions having been selected. Other than that, it can lead to globally suboptimal choices, because it is a local optimization. The fact that it is an entirely local transformation, however, makes it easy to implement. Its limit is that it has only a very narrow and local view of the program.

Postoptimizations can be fairly crucial. A strong set of postoptimizations can in fact lead to a lot more than 10% performance improvement. A few compilers only use postoptimizations (e.g., lcc). Unfortunately, postoptimizations are often quite processor dependent and not very systematic.

11 Summary

There is a very large number of optimization techniques for compiler design (more than 20 standard techniques, not counting special purpose optimizations). One might think that the fastest code would result from just using all of these optimizations. That is not the case, because an optimization that takes a long time to perform, but only has marginal effect on the execution speed is hardly worth the effort. Also one optimization technique may undo the effects achieved by another one. Unfortunately, it is a largely open problem which optimizations to use and in which order.

Several optimizations also achieve quite similar effect and their combination is not better than the parts. Except for cache optimization and strength reduction, which can have significant impact on numerical programs, a rule of thumb for arbitrary programming languages is that the first optimization yields 15%, all others less than 5%. For programming languages like C0 that give very structured guarantees (e.g., no strange side effects everywhere), this can be improved. For numerical programs, strength reduction has an impact of more than a factor of 2 and cache optimization some factor of 2 to 5.

Reasonably strong optimizing compilers can already be achieved with a small selection of optimization techniques, e.g., SSA, strength reduction, common subexpression elimination (CSE), partial redundancy elimination (PRE, to be discussed later).

Quiz

1. For every order of the optimizations you have seen so far, give an example that gets optimized to suboptimal code or explain why that does not happen.

2. For each basic optimization, give an example where the optimization makes the code run slower than unoptimized or explain why that does not happen.

3. Are there normalizing transformations on memory operations?

4. Should constant propagation, copy propagation, and CSE be implemented separately or together?

5. Should CSE be implemented separately or during SSA construction?

6. Is there a dataflow analysis that is more complicated for the basic optimizations on nonSSA than the corresponding dataflow analysis that you need to do during SSA to get the basic optimizations for free?

7. Should a compiler do postoptimizations? Or would it be a better design if it didn't?

8. Does a compiler have to worry about CISC and similar phenomena usually done in peephole optimization? Should it?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[McK65] William M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.