Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: September 14, 2018

Chapter 12 – Optimization

12.1 Overview

Using the basic code generation strategy shown in the previous chapter, you can build a compiler that will produce perfectly usable, working code. However, if you examine the output of your compiler, there are many ways in which you can see obvious inefficiencies. This stems from the fact that the basic code generation strategy considers each program element in isolation, and must use the most conservative strategy to connect them together.

In the early days of high level languages, before optimization strategies were widespread, code produced by compilers was widely seen to be inferior to that which was hand-written by humans. Today, a modern compiler has many optimization techniques and very detailed knowledge of the underlying architecture, and so compiled code is usually (but not always) superior to that written by humans.

Optimizations can be applied at multiple stages of the compiler. It's usually best to solve a problem at the highest level of abstraction possible. For example, once we have generated concrete assembly code, about the most we can do is eliminate a few redundant instructions. But, if we work with a linear IR, we can speed up a long code sequence with smart register allocation. And if we work at the level of a DAG or an AST, we can eliminate entire chunks of unused code.

Optimizations can occur at different scope within a program. **Local optimizations** refer to changes that are limited to a single basic block, which is a straight-line sequence of code without any flow control. **Global optimizations** refer to changes applied to the entire body of a function (or procedure, method, etc), consisting of a control-flow-graph where each node is a basic block. **Interprocedural optimizations** are even larger, and take into account the relationships between different functions. Generally, optimizations at larger scopes are more challenging, but have more potential to improve the program.

This chapter will give you a tour of some common code optimization techniques that you can either implement in your project compiler, or explore by implementing them by hand. But this is just an introduction: code optimization is a very large topic that could occupy a whole second text-

book, and is still an area of active research today. If this chapter appeals to you, then check out some of the more advanced books and articles referenced at the end of the chapter.

12.2 Optimization in Perspective

As a programmer – the user of a compiler – it's important to keep a sense of perspective about compiler optimizations. Most production compilers do not perform any major optimizations when run with the default options. There are several reasons for this. One reason is compilation time: when initially developing and debugging a program, you want to be able to edit, compile, and test many times rapidly. Almost all optimizations require additional compilation time, which doesn't help in the initial development phases of a program. Another reason is that not all optimizations automatically improve a program: they may cause it to use more memory, or even run longer! Yet a third reason is that optimizations can confuse debugging tools, making it difficult to relate program execution back to the source code.

Thus, if you have a program that doesn't run as fast as you would like, it's best to stop and think about your program from first principles. Two suggestions to keep in mind:

- Examine the overall algorithmic complexity of your program: a binary search ($O(\log n)$) is always going to outperform a linear search ($O(n)$) for sufficiently large n. Improving the high level approach of your program is likely to yield much greater benefits than a low-level optimization.
- Measure the performance of your program. Use a standard profiling tool like `gprof` [4] to measure where, exactly, your program spends most of its time, and then focus your efforts on improving that one piece of code by either rewriting it, or enabling the appropriate compiler optimizations.

Once your program is well-written from first principles, then it is time to think about enabling specific compiler optimizations. Most optimizations are designed to target a certain pattern of behavior of code, and so you may find it helpful to write your code in those easily-identified patterns. In fact, most of the patterns discussed below can be performed by hand without the compiler's help, allowing you to do a head-to-head comparison of different code patterns.

Of course, the fragments of code presented in this chapter are all quite small, and thus are only significant if they are performed a large number of times within a program. This typically happens inside one or more nested loops that constitute the main activity of a program, often called the **kernel** of a computation. To measure the cost of, say, multiplying two

```
#include <time.h>

struct timeval start,stop,elapsed;
gettimeofday(&start,0);

for(i=0;i<1000000;i++) {
    x = x * y;
}

gettimeofday(&stop,0);
timersub(&stop,&start,&elapsed);

printf("elapsed: %d.%06d sec",
       elapsed.tv_sec,elapsed.tv_usec);
```

Figure 12.1: Timing a Fast Operation

values together, perform it one million times within a timer interval, as shown in Figure 12.1

Careful: The timer will count not only the action in the loop, but also the code implementing the loop, so you also need to normalize the result by subtracting the runtime of an empty loop.

12.3 High Level Optimizations

12.3.1 Constant Folding

Good programming practices often encourage the liberal use of named constants throughout to clarify the purpose and meaning of values. For example, instead of writing out the obscure number `86400`, one might write out the following expression to yield the same number:

```
const int seconds_per_minute=60;
const int minutes_per_hour=60;
const int hours_per_day=24;

int seconds_per_day = seconds_per_minute
                    * minutes_per_hour
                    * hours_per_day;
```

The end result is the same (`86400`) but the code is much clearer about the purpose and origin of that number. However, if translated literally, the program would contain three excess constants, several memory lookups, and two multiplications to obtain the same result. If done in the inner loop of a complex program, this could be a significant waste. Ideally, it

```
struct expr * expr_fold( struct expr *e )
{
    expr_fold( e->left )
    expr_fold( e->right )

    if( e->left and e->right are both constants ) {

        f = expr_create( EXPR_CONSTANT );
        f->value = e->operator applied to
                   e->left->value and e->right->value
        expr_delete(e->left);
        expr_delete(e->right);

        return f;
    } else {
        return e;
    }
}
```

Figure 12.2: Constant Folding Pseudo-Code

should be possible for the programmer to be verbose without resulting in an inefficient program.

Constant folding is the technique of converting an expression (or part of an expression) combining multiple constants into a single constant. An operator node in the tree with two constant child nodes can be converted into a single node with the result of the operation computed in advance. The process can cascade up so that complex expressions may be reduced to a single constant. In effect, it moves some of the program's work from execution-time to compile-time.

This can be implemented by a recursive function that performs a post order traversal of the expression tree. Figure 12.2 gives pseudo code for constant folding on the AST.

One must be careful that the resulted computed in advance is *precisely* equal to what would have been performed at runtime. This requires using variables of the same precision and dealing with boundary cases such as underflow, overflow, and division by zero. In those cases, it is typical to force a compile-time error, rather than compute an unexpected result.

While the effects of constant folding may seem minimal, it often is the first step in enabling a chain of further optimizations.

12.3.2 *Strength Reduction*

Strength reduction is the technique of converting a special case of an expensive operation into a less expensive operation. For example, the source code expression `x^y` for exponentiation on floating point values is, in general, implemented as a call to the function `pow(x,y)`, which might be implemented as an expansion of a Taylor series. However, in the particular case of `x^2` we can substitute the expression `x*x` which accomplishes the same thing, avoiding the extra cost of a function call and many loop iterations. In a similar way, multiplication/division by any power of two can be replaced with a bitwise left/right shift, respectively. For example, `x*8` can be replaced with `x<<3`.

Some compilers also contain rules for strength reduction of operations in the standard library. For example, recent versions of `gcc` will substitute a call to `printf(s)` with a constant string `s` to the equivalent call to `puts(s)`. In this case, the strength reduction comes from reducing the amount of code that must be linked into the program: `puts` is very simple, while `printf` has a large number of features and further code dependencies.[1]

12.3.3 *Loop Unrolling*

Consider the common construct of using a loop to compute variations of a simple expression many times:

```
for(i=0;i<400;i++) {
   a[i] = i*2 + 10;
}
```

In any assembly language, this will only require a few instructions in the loop body to compute the value of `a[i]` each time. But, the instructions needed to control the loop will be a significant fraction of the execution time: each time through the loop, we must check whether `i<400` and jump back to the top of the loop.

Loop unrolling is the technique of transforming a loop into another that has fewer iterations, but does more work per iteration. The number repetitions within the loop is known as the **unrolling factor**. The example above could be safely transformed to this:

```
for(i=0;i<400;i++) {
   a[i] = i*2 + 10;
   a[i+1] = (i+1)*2 + 10;
   a[i+2] = (i+3)*2 + 10;
```

[1] While there is a logic to this sort of optimization, it does seem like an unseemly level of familiarity between the compiler and the standard library, which may have different developers and evolve independently.

```
    a[i+3] = (i+4)*2 + 10;
}
```

Or this:

```
for(i=0;i<400;i++) {
    a[i] = i*2 + 10;
    i++;
    a[i] = i*2 + 10;
    i++;
    a[i] = i*2 + 10;
    i++;
    a[i] = i*2 + 10;
}
```

Increasing the work per loop iteration saves some unnecessary evaluations of `i<400`, and it also eliminates branches from the instruction stream, which avoids pipeline stalls and other complexities within the microprocessor.

But how much should a loop be unrolled? The unrolled loop could contain 4, 8, 16 or even more items per iteration. In the extreme case, the compiler could eliminate the loop entirely and replace it with a finite sequence of statements, each with a constant value:

```
a[0] = 0 + 10;
a[1] = 2 + 10;
a[2] = 4 + 10;
. . .
```

As the unrolling factor increases, unnecessary work in the loop structures are eliminated. However, the increasing code size has its own cost: instead of reading the same instructions over and over again, the processor must keep loading new instructions from memory. If the unrolled loop results in a working set larger than the instruction cache, then performance may end *worse* than the original code.

For this reason, there are no hard-and-fast rules on when to use loop unrolling. Compilers often have global options for unrolling that can be modified by a `#pragma` placed before a specific loop. Manual experimentation may be needed to get good results for a specific program. [2]

12.3.4 Code Hoisting

Sometimes, code fragements inside a loop are constant with each iteration of the loop. In this case, it is unnecessary to recompute on every iteration,

[2] The GCC manual has this to say about the `-funroll-all-loops` option: "This usually makes programs run more slowly."

and so the code can be moved to the block preceding the loop, which is known as **code hoisting**. For example, the array index in this example is constant throughout the loop, and can be computed once before the loop body:

```
for(i=0;i<400;i++) {
   a[x*y] += i;
}
```

```
t = x*y;
for(i=0;i<400;i++) {
   a[t] += i;
}
```

Unlike loop unrolling, code hoisting is a relatively benign optimization. The only cost to the optimization is that the computed result must occupy either a temporary location for the duration of the loop, which slightly increases either register pressure or local storage consumption. This is offset by the elimination of unnecessary computation.

12.3.5 Function Inlining

Function inlining is the process of substituting a function call with the effect of that function call directly in the code. This is particularly useful for brief functions that exist to improve the clarity or modularity of code, but do not perform a large amount of computation. For example, suppose that the simple function `quadratic` is called from many times within a loop, like this:

```
int quadratic( int a, int b, int x ) {
  return a*x*x + b*x + 30;
}

for(i=0;i<1000;i++) {
    y = quadratic(10,20,i*2);
}
```

The overhead of setting up the parameters and invoking the function likely exceeds the cost of doing the handful of additions and multiplies within the function itself. By inlining the function code into the loop, we can improve the overall performance of the program.

Function inlining is most easily performed on a high level representation such as an AST or a DAG. First, the body of the function must be duplicated, then the parameters of the invocation must be substituted in. Note that, at this level of evaluation, the parameters are not necessarily constants, but may be complex expressions that contain unbound values.

For example, the invocation of `quadratic` above can be substituted with the expression `(a*x*x+b*x+30)` under the binding of `a=10`, `b=20`, and `x=i*2`. Once this substitution is performed, unbound variables such as `i` are relative to the scope where `quadratic` was called, not where it was defined. The resulting code looks like this:

```
for(i=0;i<1000;i++) {
    y = 10*(i*2)*(i*2) + 20*(i*2) + 30;
}
```

This example highlights a hidden potential cost of function inlining: an expression (like `i*2`) which was previously evaluated once and then used as a parameter to the function, is now evaluated multiple times, which could increase the cost of the expression. On the other hand, this expansion could be offset by algebraic optimizations which now have the opportunity to simplify the combination of the function with its concrete parameters. For example, constant folding applied to the above example yields this:

```
for(i=0;i<1000;i++) {
    y = 40*i*i + 40*i + 30;
}
```

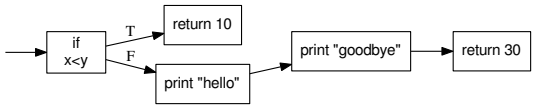
Generally speaking, function inlining is best applied to simple leaf functions that are called frequently and do little work relative to the cost of invocation. However, making this determination automatically is challenging to get right, because the benefits are relatively clear, but the costs in terms of increased code size and duplicated evaluations are not so easy to quantify. As a result, many languages offer a keyword (like `inline` in C and C++) that allow the programmer to make this determination manually.

12.3.6 Dead Code Detection and Elimination

It is not uncommon for a compiled program to contain some amount of code that is completely unreachable and will not be executed under any possible input. This could be as simple as a mistake by the programmer, who by accident returned from a function before the final statement. Or, it could be due to the application of multiple optimizations in sequence that eventually result in a branch that will never be executed. Either way, the compiler can help by flagging it for the programmer or removing it outright.

Dead code detection is typically performed on a **control flow graph** after constant folding and other expression optimizations have been performed. For example, consider the following code fragment and its control flow graph:

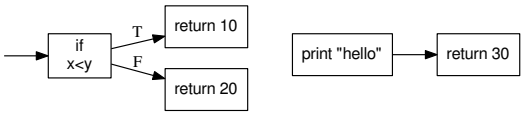
```
if( x<y ) {
   return 10;
} else {
   print "hello";
}
print "goodbye";
return 30;
```



A `return` statement causes an immediate termination of the function and (from the perspective of the control flow graph) is the end of the execution path. Here, the true branch of the `if` statement immediately returns, while the false branch falls through to the next statement. For *some* values of `x` and `y`, it is possible to reach every statement.

However, if we make a slight change, like this:

```
if( x<y ) {
   return 10;
} else {
   return 20;
}
print "goodbye";
return 30;
```



Then, both branches of the if statement terminate in a `return`, and it is not possible to reach the final `print` and `return` statements. This is (likely) a mistake by the programmer and should be flagged.

Once the control flow graph is created, determining reachability is simple: perform a traversal of the CFG, starting from the entry point of the function, marking each node as it is visited. Once the traversal is complete, if there are any nodes unmarked, then they are unreachable and the compiler may either generate a suitable error message or simply not generate code for the unreachable portion. [3]

Reachability analysis becomes particularly powerful when combined with other forms of static analysis. In the example above, suppose that variables `x` and `y` are defined as constants `100` and `200`, respectively. Constant folding can reduce `x<y` to simply `true`, with the result that the false branch of the if statement is never taken and therefore unreachable.

Now, don't get carried away with this line of thinking. If you were paying attention in your theory-of-computing course, this may sound suspiciously like the Halting Problem: can we determine whether an *arbitrary* program written in a Turing-complete language will run to completion, without actually executing it? The answer is, of course, *no, not in the general case*. Reachability analysis simply determines *in some limited cases*

[3] A slight variation on this technique can be used to evaluate whether every code path through a function results in a suitable `return` statement. This is left as an exercise to the reader.

that a certain branch of a program is impossible to take, regardless of the prorgam input. It does *not* state that the "reachable" branches of the program *will* be taken for some input, or for any input at all.

12.4 Low-Level Optimizations

All of the optimizations discussed so far can be applied to the high level structure of a program, without taking into account the particular target machine. Low-level optimizations focus more on the translation of the program structure in a way that best exploits the peculiarities of the underlying machine.

12.4.1 Peephole Optimizations

Peephole optimizations refer to any optimization that looks very narrowly at a small section of code – perhaps just two or three instructions – and makes a safe, focused change within that section. These sort of optimizations are very easy to implement as the final stage of compilation, but have a limited overall effect.

Redundant load elimination is a common peephole optimization. A sequence of expressions that both modifies and uses the same variable can easily result in two adjacent instructions that save a register into memory, and then immediately load the same value again:

Before:	**After:**
`MOVQ %R8, x` `MOVQ x, %R8`	`MOVQ %R8, x`

A slight variation is that a load to a different register can be converted into a direct move between registers, thus saving an unnecessary load and pipeline stall:

Before:	**After:**
`MOVQ %R8, x` `MOVQ x, %R9`	`MOVQ %R8, x` `MOVQ %R8, %R9`

12.4.2 Instruction Selection

In Chapter 11, we presented a simple method of code generation where each node of the AST (or DAG) was replaced with at least one instruction (and in some cases, multiple instructions). In a rich CISC instruction set, a single instruction can easily combine multiple operations, such as dereferencing a pointer, accessing memory, and performing an arithmetic operation.

To exploit these powerful instructions, we can use the technique of instruction selection by **tree coverage**. [5] The idea is to first represent each

possible instruction in the architecture as a template tree, where the leaves can be registers, constants, or memory addresses that can be substituted into an instruction.

For example, one variant of the X86 `ADDQ` instruction can add two registers together. This can be used to implement an `IADD` node in the DAG, provided the leaves of the `IADD` are stored in registers. Once the add is complete, the `ADDQ` places the result in the same register as the second argument. This is all expressed as tree fragment that matches a part of the DAG, and an instruction to be emitted, once specific register numbers are chosen:

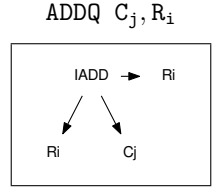


Figure 12.3 gives a few more examples of X86 instructions that can be represented as tree templates. The simple instructions at the top simply substitute one entity in the DAG for another: `MOV $Cj, Ri` converts a constant into a register, while `MOV Mx, Ri` converts a memory location into a register. Richer instructions have more structure: the complex load `MOV Cd(Rc,Ra,8), Ri` can be used to represent a combination of add, multiply, and dereference.

Of course, Figure 12.3 is not the complete X86 instruction set. To describe even a significant subset would require hundreds of entries, with multiple entries per instruction to capture the multiple variations on each instruction. (For example, you would need one template for an `ADDQ` on two registers, and another for a register-memory combination.) But this is a feasible task and perhaps easier to accomplish than hand-writing a complete code generator.

With the complete library of templates written, the job of the code generator is to examine the tree for sub-trees that match an instruction template. When one is found, the corresponding instruction is emitted (with appropriate substitutions for register numbers, etc) and the matching portion of the tree replaced with the right side of the template.

For example, suppose we wish to generate X86 code for the statement `a[i] = b + 1;` Let us suppose that `b` is a global variable, while `a` and `i` are local variables at positions 40 and 32 above the base pointer, respectively. Figure 12.4 shows the steps of tree rewriting. In each DAG, the box indicates the subtree that matches a rule in Figure 12.3

Step 1: The left `IADD` should be executed first to compute the value of the expression. Looking at our table of templates, there is no `IADD` that can directly add a memory location to a constant. So, we instead select rule (2), which emits the instruction `MOVQ b, %R0` and converts the left-hand

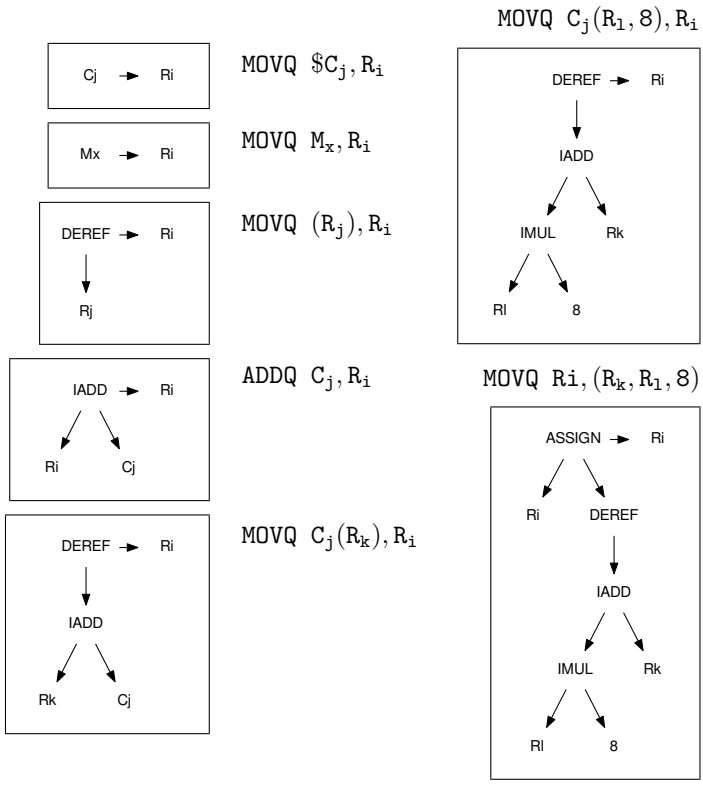


Figure 12.3: Example X86 Instruction Templates

side of the template (a memory location) into the right hand side (register %R0).

Step 2: Now we can see an `IADD` of a register and a constant, which matches the template of rule (4). We emit the instruction `ADDQ $1, %R0` and replace the `IADD` subtree with the register `%R0`.

Step 3: Now let's look at the other side of the tree. We can use rule (5) to match the entire subtree that loads the variable `i` from `%RBP+32` by emitting the instruction `MOVQ 32(%RBP), %R1` and replacing the subtree with the register `%R1`.

Step 4: In a similar way, we can use rule (6) to compute the address of `a` by emitting `LEAQ 40(%RBP), %R2`. Notice that this is, in effect, a three-address addition specialized for use with a register and a constant.

Step 5: Finally, template rule (7) matches most of what is remaining. We can emit `MOVQ R0, (R1,8)%R2` which stores the value in R1 into the computed array address of `a`. The left side of the template is replaced with the right-hand side, leaving nothing but the register `%R0`. With the tree

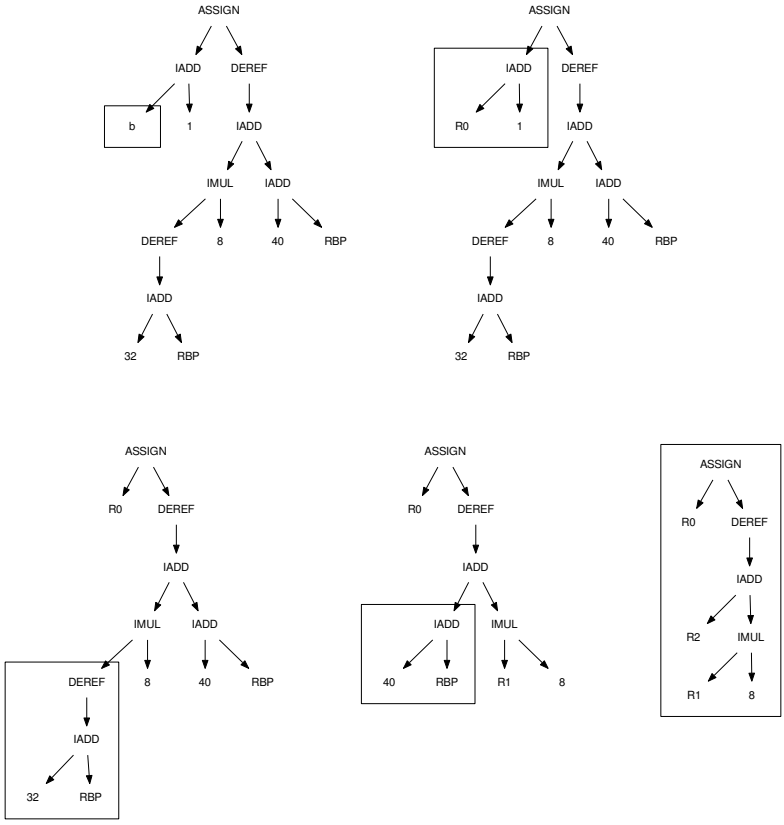


Figure 12.4: Example of Tree Rewriting

completely reduced to a single register, code generation is complete, and register `%R0` can be freed.

Under the simple code generation scheme, this 16-node DAG would have produced (at least) 16 instructions of output. By using tree coverage, we reduced the output to these five instructions:

```
MOVQ b,%R0
ADDQ $1,%R0
MOVQ 32(%RBP),%R1
LEAQ 40(%RBP),%R2
MOVQ %R0,(%R2,%R1,8)
```

12.5 Register Allocation

In modern CPUs, on-chip computational speed far outstrips memory latency: thousands of arithmetic operations can be completed in the time it takes to perform a single load of store. It follows that any optimizations that eliminates a load or store from memory can have a considerable impact on the performance of a program. Eliminating completely unnecessary code and variables is the first step towards doing this.

The next step is to assign specific local variables into registers, so they never need to be loaded from or stored to memory. Of course, there are a limited number of registers and not every variable can claim one. The process of **register allocation** is to identify the variables that are the best candidates for locating in registers instead of memory.

The mechanics of converting a variable into a register are straightforward. In each case where a value would be loaded from or stored into a memory location, the compiler simply substitutes the assigned register as the location of the value, so that it is used directly as the source or target of an instruction. The more complicated questions relate to whether it is *safe* to registerize a variable, which variables are most *important* to registerize, and which variables can coexist in registers at once. Let's look at each qeustion in turn.

12.5.1 Safety of Register Allocation

It is unsafe to registerize a variable if the eliminated memory access has some important side effect or visibility outside of the code under consideration. Examples of variables that should not be registerized include:

- Global variables shared between multiple functions or modules.
- Variables used as communication between concurrent threads.
- Variables accessed asynchronously by interrupt handlers.
- Variables used as memory-mapped I/O regions.

Note that some of these cases are more difficult to detect than others! Globally shared variables are already known to the compiler, but the other three cases are (often) not reflected in the language itself. In the C language, one can mark a variable with the `volatile` keyword to indicate that it may be changed by some method unknown to the compiler, and therefore clever optimizations should not be undertaken. Low level code found in operating systems or parallel programs is often compiled without such optimizations, so as to avoid these problems.

12.5.2 *Priority of Register Allocation*

For a small function, it may be possible to registerize *all* the variables, so that memory is not used at all. But for even a moderately complex function (or a CPU that has few available registers) it is likely that the compiler must choose a limited number of variables to registerize.

Before automatic register allocation was developed, the programmer was responsible for identifying such variables manually. For example, in early C compilers, one could add the `register` keyword to a variable declaration, forcing it to be stored in a register. This was typically done for the index variable of the inner-most loop. Of course, the programmer might not choose the best variable, or they might choose too many register variables, leaving too few available for temporaries. Today, the `register` keyword is essentially ignored by C compilers, which are capable of making informed decisions.

What strategy should we use to automatically pick variables to registerize? Naturally, those that experience the most number of loads and stores as the program runs. One could profile an execution of the program, count the memory accesses per variable, and then go back and select the top n variables. Of course, that would be a very slow and expensive procedure for optimizing a program, but one might conceivably go about it for a very performance-critical program.

A more reasonable approach would be to score variables via static analysis with some simple heuristics. In a linear sequence of code, each variable can be directly scored by the number of loads and stores it performs: the variable with the highest score is the best candidate. However, a variable access that appears inside a loop is likely to have a much higher access count. How large, we cannot say, but we can assume that a loop (and each nesting of a loop) multiplies the importance of a variable by a large constant. Multiply-nested loops increase importance in the same way.

12.5.3 *Conflicts Between Variables*

Not every variable needs a distinct register. Two variables can be assigned to the same register if their uses do not conflict. To determine this, we must first compute the **live ranges** of each variable and then construct a conflict graph. Within a basic block of a linear IR, a variable is live from its first definition until its final use. (If the same code is expressed in SSA form, then each version of a variable can be treated independently, with its own live range.)

Now, each variable with an overlapping range cannot share the same register, because they must exist independently. Conversely, two variables that do not have an overlapping live range can be assigned the same variable. We can construct a **conflict graph** where each node in the graph represents a variable, and then add edges between nodes whose live ranges overlap. Figure 12.5 gives an example of a conflict graph.

```
   Code            Live Variables
1. x = 10          x
2. y = 10          x, y
3. a = x * y       x, y, a
4. b = a * a       x, a, b
5. a = b + 30      x, a, b
6. x = x + 1       x,
7. z = x * x       x, z
8. return z;       z
```

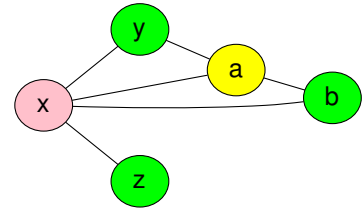


Figure 12.5: Live Ranges and Register Conflict Graph

Register allocation now becomes an instance of the **graph coloring** problem. [7] The goal is to assign each node in the graph a different color (register) such that no two adjacent nodes have the same color. A planar graph (like a two-dimensional political map) can always be colored with four colors. [4] However, a register conflict graph is not necessarily planar, and so may require a large number of colors. The general problem of finding the minimum number of colors needed is NP-complete, but there are a number of simpler heuristics that are effective in practice.

A common approach is to sort the nodes of the graph by the number of edges (conflicts), and then assign registers to the most conflicted node first. Then, proceeding down the list, assign each node a register that is not already taken by an adjacent node. If at any point, the number of available registers is exhausted, then mark that node as a non-registerized variable, and continue, because it may be still possible to assign registers to nodes with fewer conflicts.

12.5.4 Global Register Allocation

The procedure above describes the analysis of live variables and register allocation for individual basic blocks. However, if each basic block is allocated independently, it would be very difficult to combine basic blocks, because variables would be assigned to different registers, or none at all.

[4] This mathematical problem has a particularly colorful (ahem) history. In 1852, Francis Guthrie conjectured that only four colors were necessary to color a planar graph, while attempting to color a map of Europe. He brought this problem to Augustus DeMorgan, who popularized it, leading to several other mathematicians who published several (incorrect) proofs in the late 1800s. In 1891, Percy John Heawood proved that no more than *five* colors were sufficient, and that's where things stood for the next 85 years. In 1976, Kenneth Appel and Wolfgang Haken produced a computer-assisted proof of the four-color theorem, but the proof contained over 400 pages of case-by-case analysis which had to be painstakingly verified by hand. This caused consternation in the mathematical community, partly due to the practical difficulty of verifying such a result, but also because this proof was unlike any that had come before. Does it really count as a "proof" if it cannot be easily contemplated and verified by a human? [2]

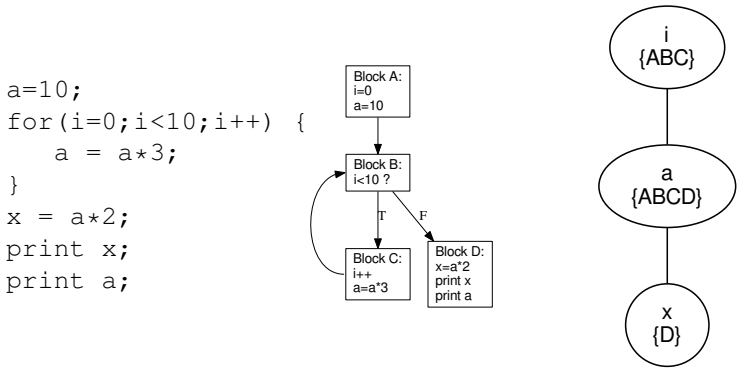


Figure 12.6: Example of Global Register Allocation
To perform register allocation on code (left) consisting of more than a basic block, build the control flow graph (middle) and then determine the blocks in which a given variable is live. Construct a conflict graph (right) such that variables sharing a live block are in conflict, and then color the graph.

It would become necessary to introduce code between each basic block to move variables between registers or to/from memory, which could defeat the benefits of allocation in the first place.

In order to perform global register allocation across an entire function body, we must do so in a way that keeps assignments consistent, no matter where the control flow leads. To do this, we first construct the control flow graph for the function. For each variable definition in the graph, trace the possible forward paths to uses of that variable, taking into account multiple paths made possible by loops and branches. If there exists a path from definition to use, then all the basic blocks in that path are members of the set of live basic blocks for that variable. Finally, a conflict graph may be constructed based on the sets of live blocks: each node represents a variable and its set of live basic blocks; each edge represents a conflict between two variables whose live sets intersect. (As above, the same analysis can be performed in SSA form for a more fine-grained register assignment.) Figure 12.6 gives an example of this analysis for a simple code fragment.

12.6 Optimization Pitfalls

Now that you have seen some common optimization techniques, you should be aware of some pitfalls common to all techniques.

Be careful of the correctness of optimizations. A given piece of code must produce the same result before and after optimization, for all possible inputs. We must be particularly careful with the boundary conditions of a given piece of code, where inputs are particularly large, or small, or run into fundamental limitations of the machine. These concerns require careful attention, particularly when applying general mathematical or logical observations to concrete code.

For example, it is tempting to apply common algebraic transformations to arithmetic expressions. We might transform `a/x + b/x` into `(a+b)/x` because these expressions are equal in the abstract world of real numbers.

Unfortunately, these two expressions are *not* the same in the concrete world of limited-precision mathematics. Suppose that `a`, `b`, and `x` are 32-bit signed integers, with a range of $[-2^31, +2^31)$. If both `a` and `b` have the value 2,000,000,000, then `a/5` is 400,000,000 and `a/5+b/5` is 800,000,000. However, `a+b` overflows a 32-bit register and wraps around to a negative value, so that `(a+b)/5` is -58993459! An optimizing compiler must be extremely cautious that any code transformations produce the same results under *all possible values* in the program.

Be careful not to change external side-effects. Many aspects of real programs depend upon the side-effects, not the results of a computation. This is most apparent in embedded systems and hardware drivers, where an operating system or an application communicates with external devices through memory-mapped registers. But, it is also the case in conventional user-mode programs that may perform I/O or other forms of communication via system calls: a `write()` system call should never be eliminated by an optimization. Unfortunately, positively identifying every external function that has side effects is impractical. An optimizing compiler must conservatively assume that any external function *might* have a side effect, and leave it untouched.

Be careful of how optimization changes debugging. A program compiled with aggressive optimizations can show surprising behavior when run under a debugger. Statements may be executed in a completely different order than stated in the program, giving the impression that the program flow jumps forwards and backwards without explanation. Entire parts of the program may be skipped entirely, if they are determined to be unreachable, or have been simplified away. Variables mentioned in the source might not exist in the executable at all. Breakpoints set on function calls may never be reached, if the function has been inlined. In short, many of the things observable by a debugger are not really program results, but hidden internal state and are not guaranteed to appear in the final executable program. If you expect your program to have bugs – and it will – better fix them before enabling optimizations.

12.7 Optimization Interactions

Multiple optimizations can interact with each other in ways that are unpredictable. Sometimes, these interactions cascade in positive ways: constant folding can support reachability analysis, resulting in dead code elimination. On the other hand, optimizations can interact in negative ways: function inlining can result in more complex expressions, resulting in less efficient register allocation. What's worse is that one set of optimizations may be very effective on one program, but counter productive on another.

A modern optimizing compiler can easily have fifty different optimization techniques of varying complexity. If it is only a question of turning each optimization on or off, then there are (only) 2^{50} combinations. But, if the optimizations can be applied in any order, there are $\mathrm{fact}(50)$ permutations! How is the user to decide which optimizations to enable?

Most production compilers define a few discrete levels of optimization. For example, `gcc` defines `-O0` as the fastest compile speed, with no optimization, `-O1` enables about thirty optimizations with modest compile-time expense and few runtime drawbacks (e.g. dead code elimination), `-O2` enables another thirty optimizations with greater compile-time expense (e.g. code hoisting), and `-O3` enables aggressive optimizations that may or may not pay off (e.g. loop unrolling.) On top of that, individual optimizations may be turned on or off manually.

But is it possible to do better with finer-grained control? A number of researchers have explored methods of finding the best combinations of optimizations, given a benchmark program that runs reasonably quickly. For example, the CHiLL [8] framework combines parallel execution of benchmarks with a high level heuristic search algorithm to prune the overall search space. Another approach is to use genetic algorithms [9] in which a representative set of configurations is iteratively evaluated, mutated, recombined, until a strong configuration emerges.

12.8 Exercises

1. To get a good sense of the performance of your machine, follow the advice of Jon Bentley [1] and write some simple benchmarks to measure these fundamental operations:
   (a) Integer arithmetic.
   (b) Floating point arithmetic.
   (c) Array element access.
   (d) A simple function call.
   (e) A memory allocation.
   (f) A system call like `open()`.
2. Obtain a standard set of benchmark codes (such as SPEC) and evaluate the effect of various optimization flags available on your favorite compiler.
3. Implement the constant folding optimization in the AST of your project compiler.
4. Identify three opportunities for strength reduction in the C-Minorlanguage, and implement them in your project compiler.
5. Implement reachability analysis in your project compiler, using either the AST or a CFG. Use this to ensure that all flow control paths through a function end in a suitable `return` statement.
6. Write code to compute and display the live ranges of all variables used in your project compiler.
7. Implement linear-scan register allocation [6] on basic blocks, based on the live ranges computed in the previous exercise.
8. Implement graph-coloring register allocation [7] on basic blocks, based on the live ranges computed in the previous exercise.

12.9 Further Reading

1. J. Bentley, "Programming Pearls", Addison-Wesley, 1999.
   A timeless book that offers the programmer a variety of strategies for evaluating the performance of a program and improving its algorithms and data structures.

2. R. Wilson, "Four Colors Suffice: How the Map Problem Was Solved", Princeton University Press, 2013.
   A history of the long winding road from the four-color conjecture in the nineteenth century to its proof by computer in the twentieth century.

3. A. Aho, M. S. Lam, R. Sethi, J. D. Ullman, "Compilers: Principles, Techniques, and Tools", 2nd edition, Pearson, 2013.
   Affectionally known as the "Dragon Book", this is an advanced and comprehensive book on optimizing compilers, and you are now ready to tackle it.

4. S. L. Graham, P. B. Kessler, and M. K. McKusick. "Gprof: A call graph execution profiler." ACM SIGPLAN Notices, volume 17, number 6, 1982. https://doi.org/10.1145/872726.806987

5. A. Aho, M. Ganapathi, and S. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", ACM Transactions on Programming Languages and Systems, volume 11, number 4, 1989. https://doi.org/10.1145/69558.75700

6. M. Poletto and V. Sarkar, "Linear Scan Register Allocation", ACM Transactions on Programming Languages and Systems, volume 21, issue 5, 1999. https://doi.org/10.1145/330249.330250

7. G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring", ACM SIGPLAN Notices, voume 17, issue 6, June 1982.
   https://doi.org/10.1145/800230.806984

8. A. Tiwari, C. Chen, J. Chame, M. Hall, J. Hollingsworth, "A Scalable Auto-tuning framework for compiler optimization", IEEE International Symposium on Parallel and Distributed Processing, 2009.
   https://doi.org/10.1109/IPDPS.2009.5161054

9. M. Stephenson, S. Amarasinghe, M. Martin, U. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning", ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.
   https://doi.org/10.1145/781131.781141