Compilers Compilers Compilers...

Fists in the Air

CPE 431
Aaron Keen

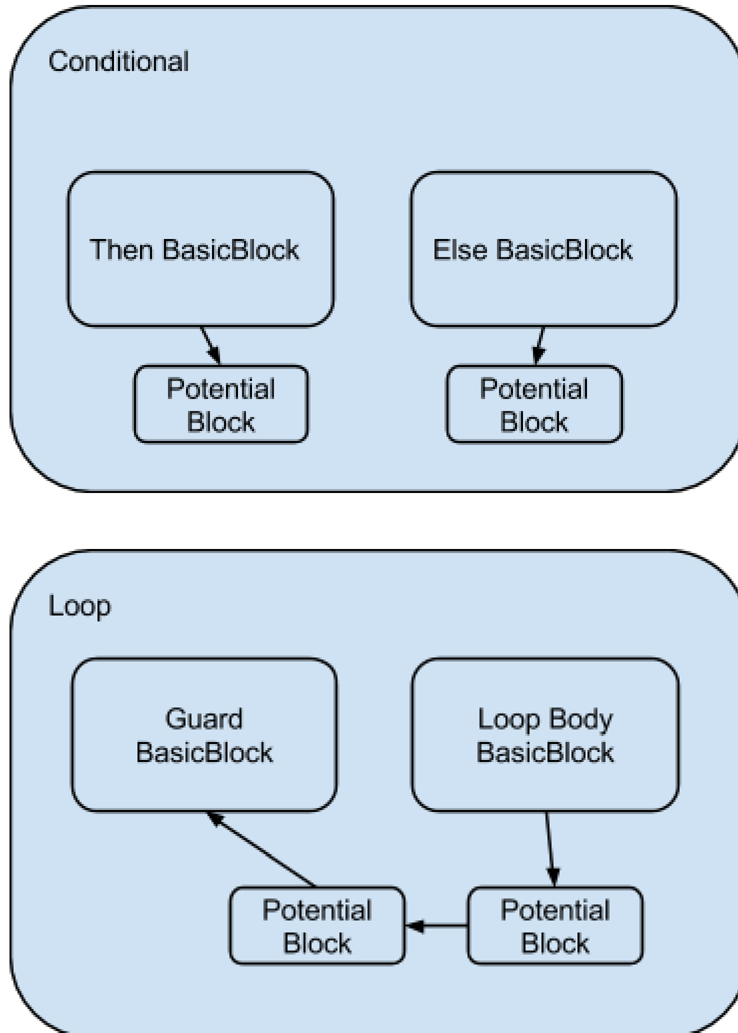
Editor’s Note

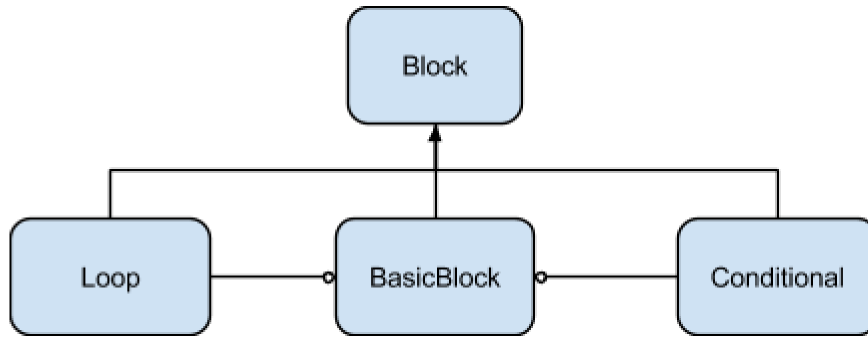
Hello Dr. Keen, it appears I have made it. It has been a sprint to the finish, and I even surprised myself on what I have accomplished. Thank you for your seemingly bottomless patience. This could be the last interaction we will have (unless I do the 4 + 1 + 1 + ...). Regardless, it has been a pleasure having you as an instructor.

Now... on to the report!

Overall Architecture

The CFG is constructed using Blocks as its foundation. A Block has exactly 1 previous Black and 1 next Block. BasicBlocks are blocks of code (no conditionals, no loops, no jumps into). Conditionals and Loops are special kinds of Blocks, where encapsulated inside each structure are BasicBlocks representing the then branch, else branch, loop guard, and loop body. The smaller components within Conditionals and Loops are not connected with each other, but taken as a whole, a Conditional and Loop contains the logic to perform the control flow.





A BasicBlock fundamentally is a list of ILOC instructions along with methods to manipulate the list of ILOC instructions such as convert them into X86 instructions and obtaining the gen set and liveOut set of this block.

Overall, the program will:

1) Perform static type checking
2) If it passes, generate a CFG
3) Perform constant folding if selected
4) Perform useless code analysis if selected
5) Display ILOC if desired
6) Generate the X86 instructions if X86 needs to be displayed
7) Display un-register-allocated X86 if desired
8) Construct an interference graph, deconstruct it, then obtain a mapping of Virtual Registers to Real Registers on an entire function
9) Map all Virtual Registers to Real Registers
10) Display X86

Data Structures

There are 2 ways of representing Instructions and Registers. On the ILOC version, instructions are ILOC instructions and registers are virtual registers. On the X86 version, instructions are X86 instructions and registers are real registers. Through a magnificent refactoring effort, VirtualRegister and RealRegister are subclasses of Register, and ILOC and X86 are subclasses of Instruction.

Each ILOC and X86 instruction has its own Java class, and each ILOC instruction has a method to convert the ILOC representation into a list of X86 instructions. All Instruction classes also have a method to return source registers and destination registers.

Optimizations Implemented

Only 2 optimizations were implemented: useless code removal by critical code analysis and constant folding.

Useless Code Removal

Useless Code Removal operates on the ILOC instructions. The general concept is that if instructions or registers are not used in critical instructions, then they are not needed and can be deleted. Critical instructions include function calls, loading in and storing out arguments, print statements, return statements, and arithmetic that lead up to compute the values for critical instructions.

Critical instructions are denoted by a boolean flag inside each ILOC Java class. Then a listing of critical sources are obtained by the instructions “getSourceRegisters” method.

Reaching for definitions is the hardest part. First, the definitions must be searched for within the current BasicBlock on the lines before the critical instruction. If none is found, it must look so its predecessors. Each Block has a single predecessor. If a Block’s predecessor is a Conditional, then that Block’s predecessor is the Conditional Block, not the 2 BasicBlock chains that form a Conditional.

After identifying which instructions are needed, they are marked as needed. Then a sweep phase removes all instructions which are not marked.

Constant Folding

Constant Folding is taking some computations away from runtime and into the compiler. It involves constants and an operator, whose evaluation is another constant. For example 4 + 5 will always be 9, so the compiler can fold 4 + 5 into a single constant 9.

Constant folding best operates in the formation of the abstract syntax tree, but is also doable in the construction of the control flow graph. If a constant is loaded into a register, that register is marked as a constant and it will also save a reference to the instruction that loaded the constant into it. If an operator such as “+” adds 2 registers, then if both registers are marked as constants, then the optimizer will add the two constants together and insert it into program. The previous 2 instructions used to load the two smaller constants are also removed from the program.

```
fun main() int
{
   int a;
   a = 1 + 2 + 3 + 4 + 5;
   return a;
}
```

```movq $1, %r11 movq $2, %rbx movq %rbx, %r10 addq %r11, %r10 movq $3, %rbx movq %rbx, %rbx addq %r10, %rbx movq $4, %r10 movq %r10, %r10 addq %rbx, %r10 movq $5, %rbx movq %rbx, %rbx addq %r10, %rbx movq %rbx, %rbx movq %rbx, %rax jmp .main_RET```	```movq $15, %rbx movq %rbx, %rbx movq %rbx, %rax jmp .main_RET```
Constant Folding OFF	Constant Folding ON

```
   movq $1, %r11
   movq $2, %rbx
   movq %rbx, %r10
   addq %r11, %r10
   movq $3, %rbx
   movq %rbx, %rbx
   addq %r10, %rbx
   movq $4, %r10
   movq %r10, %r10
   addq %rbx, %r10
   movq $5, %rbx
   movq %rbx, %rbx
   addq %r10, %rbx
   movq %rbx, %rbx
   movq %rbx, %rax
   jmp .main_RET
```

Constant Folding OFF

```
   movq $15, %rbx
   movq %rbx, %rbx
   movq %rbx, %rax
   jmp .main_RET
```

Constant Folding ON

Due to the way an AST can be constructed, a value of `a = 1 + b + 3 + b + 5` is not easily detectable. Desirably, it should be condensed into `a = b + 9`. Even though addition is a commutative property, more analysis needs to be done to observe that `1 + b + 3 + b + 5` are all commutative within the same level of operator precedence.

In addition to constant folding, conditional guards with a constant value can be evaluated and optimize the code. If the conditional guard evaluates to a constant true or false, then an entire branch can be eliminated, eliminating code that will never be run and removing possibly jump statements.

```
fun main() int
{
   int a;
   if (4 < 6)
   {
      a = 22;
   }
   else
   {
      a = 77;
   }
   return a;
}
```

<table>
<tr>
<td>
<pre>
   movq $4, %r11
   movq $6, %r10
   movq $0, %rbx
   cmpq %r10, %r11
   movq $1, %r10
   cmovlq %r10, %rbx
   cmpq $1, %rbx
   je .l1
   jmp .l2
.l1:
   movq $22, %rbx
   movq %rbx, %r12
   jmp .l3
.l2:
   movq $77, %rbx
   movq %rbx, %r12
   jmp .l3
   jmp .l3
.l3:
   movq %r12, %rax
   jmp .main_RET
</pre>
</td>
<td>
<pre>
   movq $22, %rbx
   movq %rbx, %rbx
   movq %rbx, %rax
   jmp .main_RET
</pre>
</td>
</tr>
<tr>
<td>Constant Folding OFF</td>
<td>Constant Folding ON</td>
</tr>
</table>

Currently, constant propagation is not implemented, where local variables are given a state whether it is a constant value. It would be a simple addition for the Mov instruction, in order to propagate the status of being constant to other registers.

Global Copy Propagation

Global Copy Propagation is an optimization required for Milestone 5, but is not implemented in my compiler. You will notice a fair number of redundant moving in my generated code that would be eliminated if Global Copy Propagation was implemented.

Benchmark Checklist

Below are the list of benchmarks at a glance of what my compiler generates correct unoptimized code for:

BenchMarkishTopics	PASS	bert	PASS
biggest	PASS	binaryConverter	PASS
creativeBenchMarkName	PASS	fact_sum	PASS
Fibonacci	PASS	GeneralFunctAndOptimize	PASS
hailstone	PASS	hanoi_benchmark	FAIL
killerBubbles	PASS	mile1	PASS
mixed	PASS	OptimizationBenchmark	FAIL
primes	PASS	programBreaker	PASS
stats	PASS	TicTac	PASS
uncreativeBenchmark	PASS	wasteOfCycles	PASS

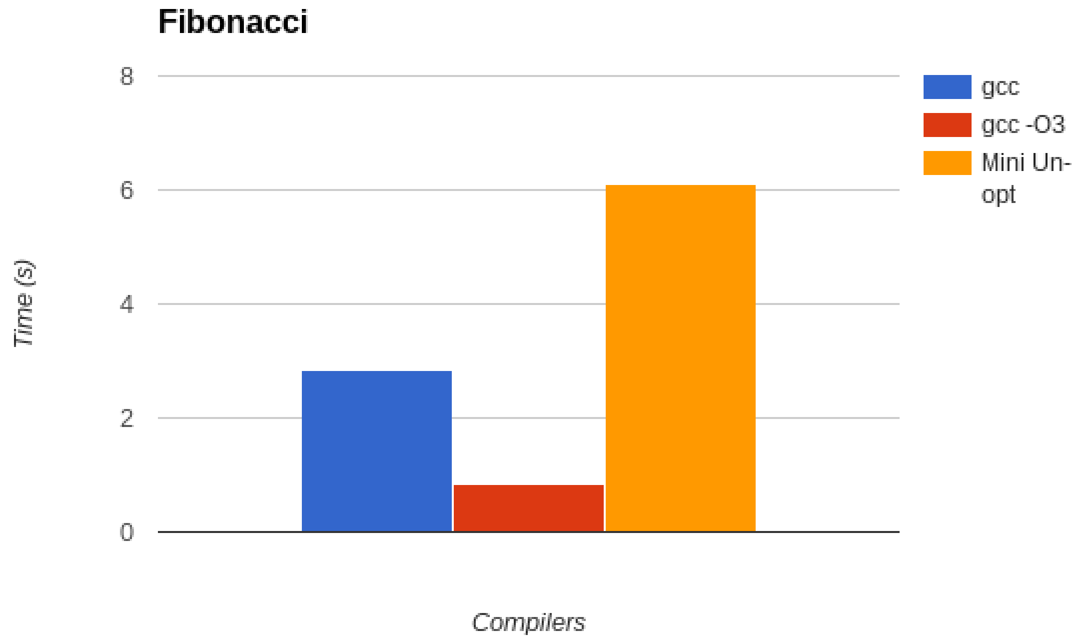
*hanoi_benchmark and OptimizationBenchmark are marked fail my compiler cannot spill those registers. Bert also has register spilling, but I can do that one.

Performance

~~There is only a small set of benchmarks that both my optimizations work on. For this test, we will look at Fibonacci and wasteOfCycles.~~

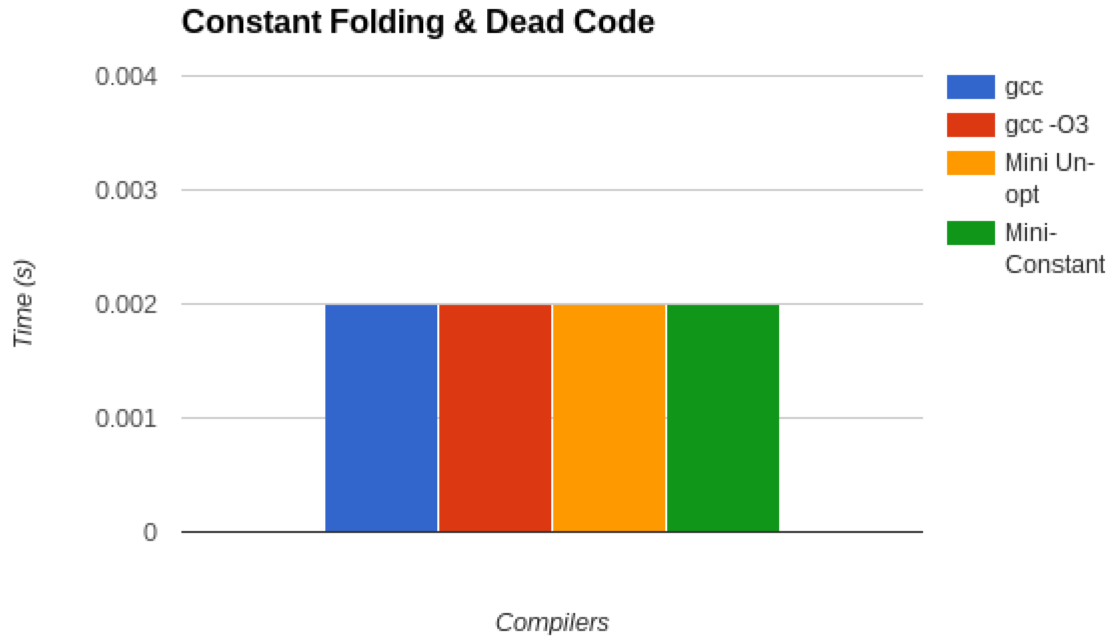
Only constant folding is a working optimization at this time.

The first benchmark to explore is Fibonacci. It will be compiled using gcc, both optimized and unoptimized, then with my compiler, only unoptimized. There are no constant folding opportunities in Fibonacci to run it optimized. Below is a chart displaying the relative effectiveness of gcc.



My compiler performs terribly! I will suggest that is because of the memory IO operations on pushing and popping the stack. Then all of the moves and copies adds to the time. In terms of program space, my executable runs at 8060 bytes and gcc -O3 runs at 8075 bytes.

A better feel of my compiler can be drawn from snippets of OptimizationBenchmark. The entire program will spill if I compile the entire thing, so I chopped it down to run only 2 functions: constant folding and dead code elimination.



The times are so small that it is hard to measure. But a better way to measure fast programs is by program size. My optimized executable runs at 7144 bytes and gcc -O3 runs at 6920 bytes. My unoptimized executable runs at 7464 bytes and gcc runs at 7080 bytes. I was hoping that useless code removal would work here, and it does! If you take a peek at the .s file generated by my compiler with useless code removal, then you will see that DeadCodeElimination does shrink! But as an odd side effect, the return of constant folding does not make it all the way to the printf. (You can see it as %rax moves to %rbx, then %r10 moves to %rsi for the printf.) The return of DeadCodeElimination does work however.

One interesting feature I noticed was for gcc, it detected useless global saves, whereas mines didn't.

<table>
<tr><td><pre>movq $9, EV_global1(%rip)</pre></td><td><pre>movq %rbx, global1
movq $5, %rbx
movq %rbx, global1
movq $9, %rbx
movq %rbx, global1</pre></td></tr>
<tr><td>gcc -O3</td><td>Useless Code Removal</td></tr>
</table>

Issues

This section will bring to light some of the design choices I made out of necessity or choice, and how they negatively impact the generated code or the architecture of the program.

Calling Convention

The calling convention requires that the caller save and restore any callee saved registers and the callee save and restore any caller saved registers. At this stage, instead of finely picking out which of the registers are alive at the time of the call, all registers are saved onto the stack. For each call, 7 registers must be pushed, then 7 registers must be popped, resulting in 14 memory operations per function call. Then for each function the user defined, 9 registers must be pushed and popped. This adds up to a lot of instructions, program space, and IO operations.

Usable Registers

The number of usable free registers are a limited set. There are 16 registers available for use, but many of them are reserved for special purposes. What I determined as the set of “free” unreserved registers are `rbx, r10, r11, r12, r13, r14, r15`. The others are used for parameter passing, stack management, and the return code. According to my compiler, only 3 benchmarks found these 7 registers insufficient enough to operate and require spilling.

Copy Propagation

Copy Propagation would eliminate what I could possibly estimate as 30 or 40 percent of useless moves. Really easy ones such as moving from a register to the same register could easily be removed, but those efforts would be saved for copy propagation.

Critical Code Analysis

Critical Code Analysis is buggy for complex programs. It requires references to its previous. By the way I structured my control flow graph, sometimes a block won’t have a previous! So I had to make some workarounds to ensure that some reaching definitions got detected, but it might not be perfect. For programs with a single block, it could be done.

References

Throughout this program, I’ve been repeatedly needed to reference the object that claims ownership of me. For example, from a Register, which instructions use me? Or from an instruction, which block do I belong in? What are my neighboring instructions? Even an instruction linked list within an instruction would be nice. In addition, I’ve thought of the idea to relax the condition that a Block must have 1 (or zero) previous Block and next Block. Maybe I could have a collection of nexts and previouses.

Conclusion

Compilers are not easy. However, just like Programming Languages, being my first time making a compiler, mistakes were made. (One which I spent a while on was accessing fields of a struct. In particular the conversion from a struct member string to an index was hard because the information was somewhere else!) My compiler performs okay, but the sprinting progress in my opinion was surprising even for me. I am now a better assembly debugger and a better programmer because of compilers.