Implementation and comparison of register allocation to translate `LLVM--` to x86

William Welle Tange

2023

Contents

1 Introduction **3**

2 Control Flow Analysis **3**
2.1 Building a graph 4
2.2 Parameterized over individual instructions 4
2.3 Parameterized over basic blocks 4

3 Liveness Analysis **5**
3.1 Dataflow Analysis 6
3.2 Constructing an interference graph 6

4 Graph Coloring **6**
4.1 Coloring by simplification 7
4.2 Coalescing 7

5 Linear Scan **7**

6 Instruction Selection **7**
6.1 `LLVM--` instruction set 7
6.2 Translating to x86 8

7 Further Optimization **8**

8 Evaluation **8**
8.1 Benchmarking 8
8.1.1 Measurements 9
8.2 Comparison to other work and ideas for future work 10

9 Conclusion **10**

A LLVM– instruction set **11**

B Benchmarks **11**

1 Introduction

Compilation refers to the process of translating from one language to another, most often from a high-level programming language intended for humans to work with, to machine- or bytecode intended to be executed on a target architecture. This process can be divided into several distinct phases, which are grouped into one of two stages colloquially referred to as the *frontend* and *backend*, the former translating a high-level programming language to an *intermediate representation* (IR) and the latter translating IR to executable machine code of a target architecture or bytecode of a target *virtual machine* (VM).

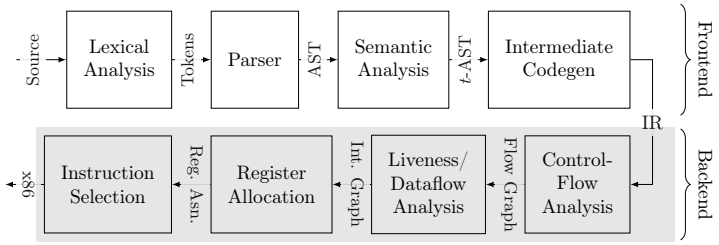

Figure 1: Compiler phases, backend highlighted

Most operations of a general-purpose programming language are translated to a set of control, logic, and arithmetic instructions to be executed sequentially on a computer processor: a single circuit/chip, referred to as the *central processing unit* (CPU), the design of which has varied and evolved over time.

Most CPUs are *register machines*, in that they use a limited set of *general-purpose registers* (GPRs) to store working values in combination with *random access memory* (RAM) for mid-term, and other I/O peripherals for long-term storage. This can largely be attributed to performance, as register machines routinely outperform *stack machines* (**ShiYunhe2008VmsS**) that are often used in VMs. Because of the limited amount of GPRs available simultaneously, a crucial part of the backend stage for an optimizing compiler is assigning each variable of the source program to a GPR in such a way that maximizes performance without sacrificing correctness.

The process of assigning each variable to a GPR is referred to as *register allocation*, and can be approached in several different ways. This paper will seek to implement graph coloring and linear scan and evaluate them in terms of runtime performance after compilation. The primary sources will be *Modern Compiler Implementation in ML* (**tiger**) and *Compilers: Principles, techniques, and tools* (**dragon**), in addition to publications concerning the linear scan approach.

2 Control Flow Analysis

The backend of a compiler takes some form of IR as input, usually a linear sequence of instructions for each separate function. This representation is close

to the level of an actual processor by design, but it isn't immediately useful for the further analysis steps needed to generate optimized code for the target architecture. The control flow of a given program refers to the order in which instructions are executed. While the flow of most instructions is linear, in the sense that the next instruction executed is located immediately after, some transfer the flow of execution elsewhere or even terminate it.

A continuous flow of instructions is referred to as a *basic block*, defined as a sequence of instructions with no branches in or out except for the first instruction (referred to as a *leader*, immediately following either the function entry or label) and the last (referred to as a *terminator*, as it either terminates or transfers the flow of execution).

Basic blocks represent a single node in a *control flow graph* (CFG), which is a directed graph whose edges denote transfer of control flow. The unconditional branch terminator always transfers control flow to the block labelled, meaning only one successor will follow, whereas conditional branching could transfer to either of the two, but because control flow analysis is not concerned with data, it simply adds both as successors. While each block has 0-2 immediate successors, the amount of predecessors is unbounded as the amount of branches targeting a specific leader is unlimited.

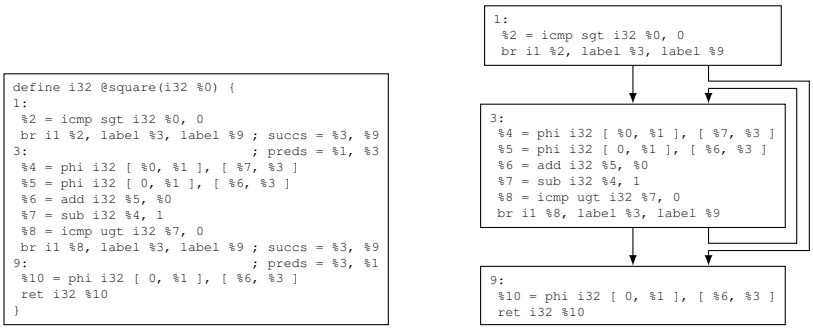

Figure 2: CFG of `@square` of `square.ll`

2.1 Building a graph

2.2 Parameterized over individual instructions

2.3 Parameterized over basic blocks

With an input stream of instructions

Each function defined in an LLVM program is constructed with the help of

3 Liveness Analysis

Translating IR with an unbounded number of variables to a CPU with a bounded number of registers involves the process of assigning each variable a register such that no value that can be used in the future is overwritten. Variables that are in use at a given program point are considered *'live'*, and although variables can be assigned the same register, variables that are live at the same time (i.e. at intersecting program points) cannot, in which case they are also said to be in interference with one another. Variables that are not in interference can be assigned the same register, and finding the precise points at which any variable is live is trivial for linear sequences of instructions. However, when conditional branching is introduced, deriving the path of execution becomes undecidable because of the halting problem.

Suppose a function that calls another:

```
define i32 @countcall(i32 %x0) {
  %x1 = add i32 %x0, 1
  call ptr @subproc()
  ret i32 %x1
}
```

Because of the halting problem, static analysis cannot determine if the call to `@subproc` will return for every possible implementation. So when assigning `%x1` a register it is undecidable whether the variable must be live in the last return instruction, i.e. needs to live across the call to another function which can overwrite several registers depending on calling conventions. Because of this, any sound approach to liveness analysis will be an approximation.

A sound albeit very naive approach is to consider every variable live at every program point, such that every variable is in interference with one another, producing a fully connected interference graph. Such a heuristic is greedy in the sense that it picks an assignment known to be safe for the least amount of preprocessing work possible. However, this can be a very inefficient assignment at runtime: a number of n variables greater than k working registers causes $n - k$ variables to be spilled to memory, which can greatly reduce performance, but allows for translation in constant time.

3.1 Dataflow Analysis

Another approach, which is a much more precise approximation, is a specific variant of the dataflow analysis as described in *Modern Compiler Implementation in ML* (**tiger**) and *Compilers: Principles, Techniques, and Tools* (**dragon**). In general, dataflow analysis is the process of finding the possible paths in which data may propagate through different branches of execution. While several applications of this exist (like constant propagation, reaching defnitions, available expressions etc.), one that is immediately beneficial in the case of liveness analysis is one that traverses a CFG in the reverse order of execution (i.e. *backwards* flow), and extracts any variable that *may* be used in execution (also referred to as *backwards may* analysis).

This algorithm calculates which program points each variable may be accessed from with some conservative constraints known to maintain correctness. Specifically, these are the transfer and control-flow constraints.

The transfer constraint is based on a *transfer function* that describes how liveness is affected across instructions. For each instruction, there is a transfer function that describes how liveness changes from one point to the one immediately after. For example, as an arithmetic operation needs to be assigned a new temporary variable, the liveness of a new variable is propagated to all instructions executed subsequently. This is done by applying the transfer function to the current *live-out* set to stop further propagation of variables defined by the currently visited instruction.

$$in\,[n] = use\,[n] \cup (out\,[n] - def\,[n]) \tag{1}$$

Control-flow constraints on the other hand propagate the use of variables to previously executed instructions, expecting these to be defined somewhere further up the CFG. This is done by propagating the union of the *live-in* set associated with all immediate successor nodes. This is also referred to as the *meet operator*, whose operator depends on the type of dataflow alanysis as well, but for liveness analysis a union is performed on the previos *live-in* variables.

$$out\,[n] = \bigcup_{s \in succ[n]} in\,[s] \tag{2}$$

Initially, two sets are associated with each instruction: the *live-in* and *live-out* sets, which are the sets of variables that are live respectively before and after execution. Then the following two equations are applied iteratively until a fixed point is reached, i.e. a point in which neither $in[n]$ or $out[n]$ is changed for all n instructions.

3.2 Constructing an interference graph

4 Graph Coloring

The heuristic for graph coloring as introduced in Appel and Aho et al. texts is a linear time approximation of an NP-complete problem. It is based on an

iterative approach.

Let k be the number of working registers available on the target architecture. After building the interference graph G, a node n with fewer than k neighbors is chosen. As n has at most $k-1$ neighbors, it can be removed safely by ensuring it is not assigned any color assigned to its neighbors, effectively simplifying G.

It must hold that $G - \{n\}$ is k-colorable if G is k-colorable

4.1 Coloring by simplification

4.2 Coalescing

Coalescing is the process of eliminating moves/copies of data from one GPR to another by combining their interference graph nodes.

5 Linear Scan

As linear scan

6 Instruction Selection

6.1 `LLVM--` instruction set

The intermediate representation emitted by the frontend of a compiler serves as a stepping stone independent of the target architecture. The `LLVM` infrastructure is the industry standard in terms of bridging this gap and was consequently the library used to translate the semantically annotated abstract syntax tree to executable machine code in the 2022 compilers course. As this project is an expansion on this, it follows naturally to build on this.

The instruction set used in this paper will be a union of the sets used in the 2022 and 2023 compilers courses in order to work as a drop-in replacement of LLVM for either of the two respective source languages: Tiger and Dolphin. This instruction set is a subset of the one used in practice, as, for instance, neither of the languages implemented support exception handling, floating point operations and so on, and instead only strive to cover the basics of compilers.

The instructions included are `trunc` has only been added in order to help cover more generated LLVM. The branching for most of these is trivial: after successful execution the flow of all but `br`, `ret` and `unreachable` will unconditionally attempt to execute the next instruction in memory (i.e. increment the instruction pointer by instruction length).

Because of this, these instructions are referred to as *terminators*, as their purpose is to disrupt what had otherwise been a linear flow from the beginning of this continus sequence of instructions, henceforth referred to as a *basic block*.

6.2 Translating to x86

Translating each IR instruction to x86 correctly is a matter of eliminating unintended side-effects. Each LLVM– instruction is defined to have only one purpose, as concepts such as calling conventions, stack frames, or a `FLAGS` register are completely abstracted over in order to remain platform independent.

In contrast, the x86 *instruction set architecture* (ISA) is targeting a *complex instruction set computer* (CISC) family of processors, the instructions of which perform a much broader set of operations **tiger**. This is is in part due to pipelining, i.e. an abstraction over the concrete implementation of the actual processor, which in turn is made for the sake of performance.

An example of this would be the division/remainder operation: since integer division is a non-trivial iterative process wherein both the quotient and remainder is needed throughout, the result of both of these is stored in the `%rax` and `%rdx` registers respectively. This means two operations are performed simultaneously regardless of which value is used, hence they need to be restored before executing the next instruction, as any variables assigned to `%rax` or `%rdx` will be overwritten.

add	addx	{}
mul	imul	{ %rax }
sdiv	idivx	{%rax, %rdx}
srem	idivx	{%rax, %rdx
call	callx	{%rax, *caller saved registers*}

7 Further Optimization

8 Evaluation

Relevant metrics by which to compare these variations are naturally the performance of the code generated at runtime, but also the time efficiency at compile-time. While the code written isn't expected to outperform LLVM, due to the time complexity of the linear scan and simplification algorithms implemented their respective runtime performances are expected to outperform the builtin graph coloring algorithm.

Another factor worth considering would be memory usage, caches misses, garbage collection (although not relevant to this as it doesn't use GC), vectorizatiion (loop optimizations by SIMD) etc.

8.1 Benchmarking

There are different approaches to assessing how well a compiler backend translates IR to machine code targeting a specific architecture and platform. One of the primary means is to simply measure the time it takes to execute the code generated by it.

Since the target architecture is x86-64, all benchmarks are executed natively on the fastest processor available to be used consistently at the time of writing. This is to maximize the sample size and in turn reduce uncertainty, but the clock speed at which the benchmarks are performed is irrelevant to how fast the generated code is. Because of this, the metric recorded is the amount of CPU cycles spent executing from start to finish instead of actual time in seconds, as the time measure only works as a more imprecise estimate of the amount of underlying work/execution steps performed on the processor.

Needless to say, CPU cycles isn't a perfect measurement either, but it works to eliminate the clock speed as well as negate much of the time the scheduler allocates for other processes. Scheduling still has a negative impact on execution because of the cache misses caused by context switching, but outside of executing each program in immediate mode which isn't possible on Linux or macOS this is a sensible estimate. To further reduce context switching, benchmarks are made on a fresh restart with minimal background processes running.

Additionally, all benchmarks are deterministic, so the value of each virtual variable will stay the same at every instruction step of execution when compared to another run with the same starting conditions. This means that the cycles measured over infinitely many runs will converge towards a 'perfect' run with no cache misses as caused by context switching. So across n runs, the run least affected by context switching will be the one with the least CPU cycles, so is the one most representative of the actual performance without noise.

Measurements are made with the `perf` utility, which relies on the *hardware counters* (HC) registers of modern microprocessors, which are special purpose registers meant to record performance data live during execution. Because the analysis is integrated into the chip on which it is executing, very little overhead is incurred, and is therefore the go-to for estimating native performance.

8.1.1 Measurements

Each `.ll` file present in the `benches` directory is listed here in a table denoting the input paramater on the left column and type of allocator used.

8.1.1.1 fib.ll

The 'dumb' fib is a very naive approach to finding the nth number of the Fibonacci sequence by calling itself recursively twice (decrementing n before each), which causes an exponential growth in function calls.

fib	clang	greedy	simple	linear
40	1.000000	1.438693	1.441270	0.73
41	1.000000	1.441035	1.440626	0.73
42	1.000000	1.447140	1.446094	0

8.2 Comparison to other work and ideas for future work

9 Conclusion

A LLVM– instruction set

1. binary operations `add`, `and`, `ashr`, `lshr`, `mul`, `or`, `sdiv`, `srem`, `shl`, `sub` and `xor`
2. integer comparison `icmp` (conditions being `eq`, `ne`, `sge`, `sgt`, `sle` and `slt`)
3. memory/address operations `alloca`, `gep`, `load` and `store`
4. `mov` operations `gep`, `phi`, `ptrtoint`, `trunc`, and `zext`
5. control flow operations `br`, `call` and `ret`
6. a nullary block terminator `unreachable`

B Benchmarks

B.1 `benches/fib.ll`

```
declare i32 @atoi(i8*)
define i32 @fib(i32 %n0) {
  %cn =  icmp sle i32 %n0, 2
  br i1 %cn, label %base, label %rec
base:
  ret i32 1
rec:
  %n1 = sub i32 %n0, 1
  %v0 = call i32 @fib(i32 %n1)
  %n2 = sub i32 %n0, 2
  %v1 = call i32 @fib(i32 %n2)
  %v2 = add i32 %v1, %v2
  ret i32 %v2
}
define i32 @main(i32 %argc, i8** %argv) {
  %arg1ptr = getelementptr i8*, i8** %argv, i64 1
  %arg1 = load i8*, i8** %arg1ptr
  %n = call i32 @atoi(i8* %arg1)
  call i32 @fib(i32 %n)
  ret i32 0
}
```

B.1.1 `make fib-clang`

```
$ make fib-clang
clang -O0 -target x86_64-unknown-darwin benches/fib.ll -o fib-clang
```

B.1.1.1 `make bench fib-clang 42`

```
$ make bench fib-clang 42
/usr/bin/time -al ./fib-clang 42
        1.58 real         1.56 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 765  page reclaims
                   0  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                  29  involuntary context switches
         11000239251  instructions retired
          4962802652  cycles elapsed
             1708928  peak memory footprint
```

B.1.1.2 `make bench fib-clang 43`

```
make bench fib-clang 43
$ /usr/bin/time -al ./fib-clang 43
        2.71 real         2.49 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 700  page reclaims
                  66  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                  13  voluntary context switches
                  32  involuntary context switches
         17797124508  instructions retired
          8035601598  cycles elapsed
             1708928  peak memory footprint
```

B.1.1.3 `make bench fib-clang 44`

```
$ make bench fib-clang 44
/usr/bin/time -al ./fib-clang 44
        4.06 real         4.04 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 700  page reclaims
                  66  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   5  voluntary context switches
                  29  involuntary context switches
          28781328564  instructions retired
          12988681489  cycles elapsed
             1708928  peak memory footprint
```

B.1.1.4 `make bench fib-clang 45`

```
$ make bench fib-clang 45
/usr/bin/time -al ./fib-clang 45
        6.54 real         6.51 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 700  page reclaims
                  66  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   6  voluntary context switches
                  39  involuntary context switches
          46555259189  instructions retired
          20968319385  cycles elapsed
             1708928  peak memory footprint
```

B.1.1.5 `make bench fib-clang 46`

```
$ make bench fib-clang 46
/usr/bin/time -al ./fib-clang 46
       10.58 real        10.54 user         0.00 sys
```

```
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 700  page reclaims
                  66  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   5  voluntary context switches
                 107  involuntary context switches
         75314385847  instructions retired
         33939361073  cycles elapsed
             1725376  peak memory footprint
```

B.1.1.6 `make bench fib-clang 47`

```
$ make bench fib-clang 47
/usr/bin/time -al ./fib-clang 47
       17.09 real        17.06 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                 766  page reclaims
                   0  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                 100  involuntary context switches
        121838577947  instructions retired
         54901966523  cycles elapsed
             1708928  peak memory footprint
```

B.1.1.7 `make bench fib-clang 48`

```
$ make bench fib-clang 48
/usr/bin/time -al ./fib-clang 48
       27.65 real        27.59 user         0.00 sys
             2678784  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
```

```
                  0  average unshared stack size
                766  page reclaims
                  0  page faults
                  0  swaps
                  0  block input operations
                  0  block output operations
                  0  messages sent
                  0  messages received
                  0  signals received
                  0  voluntary context switches
                161  involuntary context switches
       197130073055  instructions retired
        88835375488  cycles elapsed
            1708928  peak memory footprint
```