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

William Welle Tange

2023

Contents

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 platform. 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) layer and the latter translating IR to executable machine code on a target platform or bytecode for a target *virtual machine* (VM).
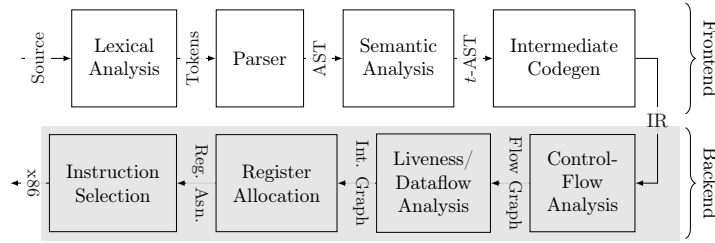

Figure 1: Compiler phases, backend highlighted

Most operations of a general-purpose programming language are translated to a set of logic, control, and arithmetic instructions that are intended to be executed sequentially on a computer processor: a single circuit/chip, referred to as the *central processing unit* (CPU). CPU design varies and has evolved over time, but most processors executing logic operations 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 so-called *stack machines* (Shi et al., 2008) 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* (Appel, 1997) and *Compilers: Principles, techniques, and tools* (Aho et al., 2014), in addition to publications concerning the linear scan approach.

2   Review of Literature

The two primary sources are the *Modern Compiler Implementation in ML* Appel, 1997 and Dragonbook Aho et al., 2014, in addition to scientific publications concerning the linear scan approach.

2.1   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 to be executed is located immediately after, some send the execution elsewhere or outright terminate it.

These contiguous sequences of instructions are referred to as *basic blocks*: a basic block is defined as a sequence of instructions with no branches in or out except for the first and last instructions, and can be collectively construed of as a node in a control flow graph, whose directed entry edges are from any block that directs control flow towards itself.

These instructions are referred to as *terminators*, as they potentially terminate a contiguous flow of execution. This subset of instructions can be further divided into *branch instructions* and those concerned with handling the callstack in function/subroutine management.

hence a *control flow graph* is derived from these. A terminator will branch elsewhere to continue execution. The destination of this has to be an annotated by a label, hence labels *initiate* blocks.

2.2   Liveness Analysis

Liveness analysis is the process of finding the program points at which a variable is live. This serves the function of determining which variables are live simultaneously, i.e. in interference/conflict with one another, which cannot be assigned the same physical register without overwriting the value of one another in execution. These can be derived recursively by iteratively applying so-called *dataflow equations* to each node in a *control-flow graph* (CFG) until a stable state/fixed point is reached.

The algorithm described in *Modern Compiler Implementation in ML* Appel, 1997 and Aho et al., 2014 is based on the following equations for the live-in and live-out variables respectively:

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

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

3

where $use[n]$ is the set of all dependent variables, $def[n]$ is the set of all variables defines and $succ[n]$ is the set of all immediate successor nodes of n.

The structure of the CFG is inherently vague as both Appel, 1997 and *Compilers : Principles, Techniques, and Tools* Aho et al., 2014, while based on the same underlying conceptions of dataflow, have different approaches to nodes of a CFG, with the former taking each individual instruction into consideration and the latter each basic block. Either of these approaches are sound

2.3 Graph Coloring

2.4 Linear Scan

3 `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*.

4 Control Flow Analysis

With an input stream of instructions
Each function defined in an LLVM program is constructed with the help of

5 Liveness Analysis

Dataflow analysis

6 Graph Coloring

The graph coloring by simplification as described by Appel

6.1 Coalescing

7 Linear Scan

8 Instruction Selection

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 Appel, 1997, p. 190. 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*}

9 Further Optimization

10 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.

10.1 Benchmarking

Benchmarks are performed on Apple Silicon (M1 Pro, i.e. ARMv8.5-A Domaine et al., 2011) despite the target instruction set being x86-64. This is possible because of Rosetta 2, which translates x86-64 to ARMv8 before execution Poletto and Sarkar, 1999. Time is measured using the `time` function of the bash shell as it is native to the chosen Docker image. As all execution relevant to the performance of register allocation is done in user mode, only the `user` metric is noted.

cell1	cell2	cell3
cell4	cell5	cell6
cell7	cell8	cell9

fib	clang	greedy	briggs	linear
40	1.000000	1.438693	1.441270	0.73
41	1.000000	1.441035	1.440626	0.73

bench clang benches/fib.ll 42 min=5494929992 (1.000000) avg=5545852373 (1.000000) max=5669289413 bench greedy benches/fib.ll 42 min=7951933724 (1.447140) avg=7969947833 (1.437101) max=8004638438 bench briggs benches/fib.ll 42 min=7946186345 (1.446094)

sha256	clang	greedy	briggs	linear
1000000	1	0.73	0.73	0.73

10.2 Comparison to other work and ideas for future work

11 Conclusion

References

Aho, A. V., lan, M. S., Sethi, R., & Ullmann, J. D. (2014). *Compilers : Principles, techniques, and tools* (2nd ed., internat. ed.). Pearson Education Limited.

Appel, A. W. (1997). *Modern compiler implementation in ml.* Cambridge University Press.

Domaine, Brandner, F., Boissinot, B., Darte, A., Dinechin, B., & Rastello, F. (2011). Computing liveness sets for ssa-form programs.

Poletto, M., & Sarkar, V. (1999). Linear scan register allocation. *ACM transactions on programming languages and systems*, *21*(5), 895–913.

Shi, Y., Casey, K., Ertl, M., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. *ACM transactions on architecture and code optimization*, *4* (4), 1–36.

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
```