Implementation and evaluation of register allocation for translating LLVM-- to x86-64

William Welle Tange, 201809080

January 13, 2024

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tellus at urna condimentum mattis. Enim facilisis gravida neque convallis. Enim facilisis gravida neque convallis a cras semper auctor. Vitae proin sagittis nisl rhoncus mattis rhoncus urna neque. Justo nec ultrices dui sapien eget mi proin. Id aliquet risus feugiat in ante. A cras semper auctor neque vitae tempus quam pellentesque nec. Nulla porttitor massa id neque aliquam vestibulum morbi. Facilisi cras fermentum odio eu. Odio pellentesque diam volutpat commodo sed egestas egestas fringilla. Amet consectetur adipiscing elit duis tristique sollicitudin nibh sit amet. Adipiscing at in tellus integer feugiat. A pellentesque sit amet porttitor. Sed elementum tempus egestas sed sed risus pretium quam. Quam viverra orci sagittis eu volutpat odio facilisis mauris sit. Arcu dictum varius duis at. Sociis natoque penatibus et magnis dis parturient montes. Varius sit amet mattis vulputate enim nulla aliquet. Ac ut consequat semper viverra. Scelerisque eleifend donec pretium vulputate sapien nec. Orci ac auctor augue mauris augue. Pulvinar mattis nunc sed blandit. Semper auctor neque vitae tempus quam pellentesque nec nam. Risus pretium quam vulputate dignissim. Dui faucibus in ornare quam viverra orci sagittis eu volutpat. Ac tortor vitae purus faucibus. Eu mi bibendum neque egestas congue quisque egestas diam in. Curabitur vitae nunc sed velit dignissim sodales ut eu sem. Arcu odio ut sem nulla.

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 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* (see Figure 1). The former is translating a high-level programming language to an *intermediate representation* (IR) and the latter is translating IR to executable machine code of a target architecture or bytecode of a target *virtual machine* (VM).

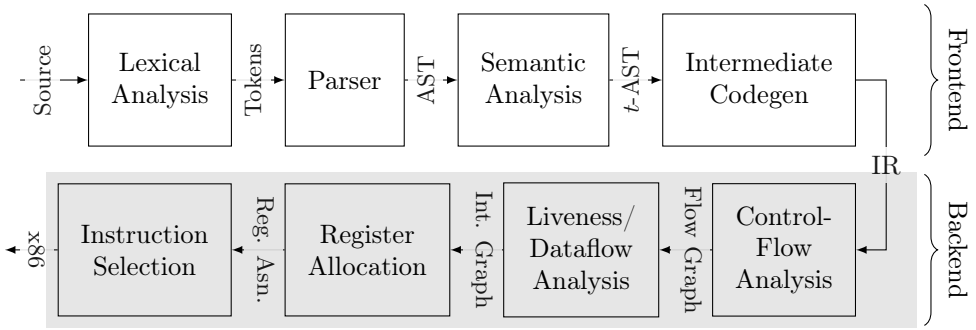


Figure 1: Compiler phases, backend highlighted

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* [1] that are often used in *virtual machines* (VMs). Although register machines generally also have a stack available, as being limited to mere bytes of storage is simply infeasible for most large scale applications, using it compared to GPRs is orders of magnitudes slower [2]. Because of this, 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 are *Modern Compiler Implementation in ML* [3] and *Compilers: Principles, techniques, and tools* [4] with regards to static analysis and graph coloring, in addition to *Linear Scan Register Allocation* [5] concerning linear scan.

The implementation is written in OCaml for the most part, although Python has been put to use to generate further tests paramterized over some value.

2 Intermediate Representation

Although compilers can feasibly translate from the source language to machine code of the target architecture directly, which could even be more efficient, doing so hinders the portability as machine code targeting architecture A isn't necessarily useful for targeting architecture B further down the line.

Instead of translating directly from a source language to target machine code, most compiler frontends emit IR which is intended as an abstraction over CPU architectures. It is by no means executable on either A or B, but it is much closer to the instruction set executed on a CPU than the source language originally fed to the frontend. This helps portability immensely, as a frontend emitting an IR no longer needs to specialize to a particular architecture, instead it can target as many as the backend supports. Likewise, a compiler backend will support any frontend provided that they emit correct IR.

Because of this, compilation of high-level programming languages will often go no further than emitting IR, then leave the rest for a subsequent backend implementation. This naturally leads to the LLVM toolchain, which is by far the most used compiler backend in practice. Most languages, if targeting native execution on a CPU, will have an LLVM implementation. This is true for C/C++ (Clang compiler [6]), D (LDC compiler [7]), Swift (official Swift compiler [8]) and Rust (official `rustc` compiler [9]) to name a few. In fact, the use of LLVM allows for them to be compiled to completely unforeseen and unintended targets, like web browsers with WebAssembly or even graphics cards with compute kernels [10].

Conversely, managed languages such as Java and C# target their own respective VMs, so they typically don't have much to gain from the LLVM toolchain. That said, some projects have attempted to implement LLVM in their translation of bytecode to native architectures, such as LLILC which promised both *just-in-time* (JIT) as well as *ahead-of-time* (AOT) compilation [11]. Similarly, Kotlin (the spiritual successor to Java) that ordinarily targets the JVM has a native backend that compiles directly to machine code using LLVM, circumventing the need for a VM entirely [12].

For example, the add function implemented in C as seen in Figure 2 is compiled to the LLVM IR as seen in Figure 3 after stripping optimization attributes with the `strip` utility. At this level of complexity they are largely equivalent, with the primary difference appearing to be syntactic. LLVM IR keeps the typed binary operations as well as the function construct with a return statement, although the variable names are discarded as IR is generally not for human interaction, except the function name, as this is used for linking with other object files where the symbol naming is used to look up offsets.

```c
int add(int a, int b) {
  int sum = a + b;
  return sum;
}
```

Figure 2: Arithmetic function implemented in C

```
define i32 @add (i32 %0, i32 %1) {
 %3 = add i32 %1, %0
 ret i32 %3
}
```

Figure 3: Stripped `clang -O1 -S -emit-llvm`

```
        .text
        .globl          add                     # -- Begin function add
add:                                            # @add
        movl            %esi, %eax
        addl            %edi, %eax
        retq
```

Figure 4: Output of `clang -S` without debug markers

When the IR is translated to x86 as seen in Figure 4, the virtual variables `%0`, `%1` and `%2` are assigned the registers `%esi`, `%edi` and `%eax` respectively. This is presumably because of several optimizations, as `%rdi` and `%rsi` are used to pass the first and second parameters and the `%rax` as the return value according to the System V AMD64 ABI [13], so by assigning the variables to those, several `movs` are saved. Either way, the assignment is sound and the instruction selection is reasonably close to the source code. There are other changes as well, as marking the target section with `.text` and the `.globl` annotation to export `add` in the symbol table which permits linking with other object files, but that isn't stricly related to register allocation.

2.1 Static Single-Assignment form

In compiler backends, *Static Single-Assignment* (SSA) form is a property that applies to some IR, including LLVM. It is a way of representing a program such that each variable is assigned only once in the scope of a function. Each use of the variable after definition then refers to the value of that single assignment, and any operations are applied by assigning the result of said operation to a freshly defined variable.

This property eliminates redefinitions entirely, making it easier to reason about data flow and in turn also speed up analysis. It's important to note that SSA form itself doesn't inherently mandate immutability,

despite only allowing for single assignment. Rather, it provides two approaches to mutable variables: the phi nodes and load/store operations.

Building on the earlier example, consider multiplication as implemented in Figure 5. Translated to LLVM IR with no optimization can be seen in Figure 7, which achieves mutability by allocating a variable on the stack (lines 2-5), loading the value on use and storing on reassignment. This means new values are loaded and assigned upon reentering a block, without reassigning the value of the source variable, which points to whatever slot was available at the time of definition.

```c
extern int add(int a, int b);

int mul(int a, int b) {
  int product = 0;
  for (int i = 0; i < a; i++) {
    product = add(product, b);
  }
  return product;
}
```

Figure 5: Multiplication function implemented in C

```
declare i32 @add(i32, i32)

define i32 @mul (i32 %0, i32 %1) {
 %3 = icmp sgt i32 %0, 0
 br i1 %3, label %6, label %4
4:
 %5 = phi i32 [0, %2], [%9, %6]
 ret i32 %5
6:
 %7 = phi i32 [%10, %6], [0, %2]
 %8 = phi i32 [%9, %6], [0, %2]
 %9 = call i32 @add (i32 %8, i32 %1)
 %10 = add i32 %7, 1
 %11 = icmp eq i32 %10, %0
 br i1 %11, label %4, label %6
}
```

Figure 6: Stripped `clang -O1 -S -emit-llvm`

```
define i32 @mul (i32 %0, i32 %1) {
 %3 = alloca i32
 %4 = alloca i32
 %5 = alloca i32
 %6 = alloca i32
 store i32 %0, i32* %3
 store i32 %1, i32* %4
 store i32 0, i32* %5
 store i32 0, i32* %6
 br label %7
7:
 %8 = load i32, i32* %6
 %9 = load i32, i32* %3
 %10 = icmp slt i32 %8, %9
 br i1 %10, label %11, label %18
11:
 %12 = load i32, i32* %5
 %13 = load i32, i32* %4
 %14 = call i32 @add (i32 %12, i32 %13)
 store i32 %14, i32* %5
 br label %15
15:
 %16 = load i32, i32* %6
 %17 = add i32 %16, 1
 store i32 %17, i32* %6
 br label %7
18:
 %19 = load i32, i32* %5
 ret i32 %19
}
```

Figure 7: Stripped `clang -O0 -S -emit-llvm`

Another approach is that of Figure 6, which uses so-called 'phi nodes' instead. These are a type of instruction that evaluates to a specified operand depending on which predecessor block was executed immediately prior. This circumvents the need for the same level of mutability as is achieved by reading/writing to memory, without violating the properties of SSA, by copying a value from the end of a predecessor to the beginning of the current block. A phi node represents a point in the program where the control flow merges, and it selects the appropriate version of a variable based on the path taken. This ensures that the data flow is well-defined and allows for easy analysis across different control flow paths.

3 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 outright 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 within it) and the last (referred to as a *terminator*, as it either terminates or transfers the flow of execution). In order to represent the order in which instructions are executed, a *control flow graph* (CFG) is introduced. It is a directed graph whose edges represent transfer of control. The type of node varies over the source material, with CFGs of the Appel text [3] constructed over individual instructions and the Aho text [4] over basic blocks. This project has opted to only use the Appel approach. One of the benefits is greater precision, in that several variables may be used in the same block and be assigned the same register as long as the two are not in interference within it. This wouldn't be possible with the Aho approach which completely discounts which parts of the block it is used in. Either way, leaders will have a set of predecessors and terminators a set of successors associated with them.

Terminators have one or two successors in the case of branching or none at all in the case of function exit. Unconditional branching always transfers control to the block labelled, meaning only one successor follow, whereas conditional branching could transfer to either of the two, but because control flow analysis is not concerned with data, both are considered as possible successors. Successors of node n are denoted $succ[n]$, and while each node also has a set of predecessors associated, which consists of an unbounded number of nodes from which control may be transferred, it isn't particularly useful in the following analysis.

3.1 Building a Control Flow Graph

Building a graph is relatively straight forward, with the input for every function declared parsed as a tuple type named `cfg` of the form:

```
type cfg = (lbl option * block) * (lbl * block) list
```

which consists of an optionally named entry block in the first part and a list of trailing blocks that are always named in the second. The reason for the first block to be optionally named is the fact that *some* LLVM programs transfer control back to the point of entry, whereas every subsequent block needs to be named because branching can only target it by referring to its name. Strictly speaking, this isn't how labels are treated in LLVM. Instead each block is always named, if not explicitly then by the next unused value from the same counter as temporaries [14], but this was discovered too late in development to account for.

From this, a CFG is constructed using the `OCamlgraph` library [15], which provides several approaches with various structures, but the one used in this case is `Imperative.Digraph.Abstract` where the index is parameterized over `int` which corresponds to the index of the instruction starting at 0 for the first instruction to be executed. At first, a graph is constructed with the input flattened such that labels, instructions and terminators are represented as a continuous sequence as opposed to the basic blocks that is parsed from the input.

```
let flatten ((head, tail) : Ll.cfg) : insn list =
  let label l = Label l and insn i = Insn i in
  let block (b : Ll.block) = List.map insn b.insns @ [ Term b.terminator ] in
  let named_opt (n, b) = (Option.map label n |> Option.to_list) @ block b in
  let named (n, b) = Label n :: block b in
  named_opt head @ (List.map named tail |> List.flatten)
```

This is done to simplify lookup of which graph vertex corresponds to the nth instruction/terminator. Labels, while present in the flattened list, aren't considered as they have no inherent function other than labelling the leader instruction to branch to.

```
let graph ((head, tail) : Ll.cfg) : G.V.t array * G.t =
  let insns = flatten (head, tail) |> List.mapi (fun i n -> (i, n)) in
```

```
  let verts = Array.init (List.length insns) G.V.create in
  let addbl t = function i, Label l -> S.ST.add l verts.(i + 1) t | _ -> t in
  let blocks = List.fold_left addbl S.ST.empty insns in
  let g = G.create () in
  Array.iter (G.add_vertex g) verts;
  let addterm i = function
    | Ll.Ret _ | Unreachable -> ()
    | Br lbl -> G.add_edge g verts.(i) (S.ST.find lbl blocks)
    | Cbr (_, l1, l2) ->
        G.add_edge g verts.(i) (S.ST.find l1 blocks);
        G.add_edge g verts.(i) (S.ST.find l2 blocks)
    | Switch (_, _, _, lbls) ->
        let vert l = S.ST.find l blocks in
        List.map snd lbls |> List.map vert |> List.iter (G.add_edge g verts.(i))
  in
  let addedges = function
    | _, Label _ -> ()
    | i, Insn _ -> G.add_edge g verts.(i) verts.(i + 1)
    | i, Term term -> addterm i term
  in
  List.iter addedges insns;
  (verts, g)
```

As can be seen above, instructions always have one edge to the instruction/terminator immediately following, while terminators can have any number of edges (0 for `ret` and `unreachable`, 1-2 for `br` and an unbounded number for `switch`).

4 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 may be needed 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.

For instance, suppose a function that calls another:

```
1 define i32 @countcall(i32 %x0) {
2   %x1 = add i32 %x0, 1
3   call void @printInt(i32 %x1)
4   call ptr @subproc()
5   ret i32 %x1
6 }
```

Figure 8: Increment, print and possibly return argument

Because of the halting problem, static analysis cannot necessarily determine if the call to `@subproc` will return. So when assigning `%x1` a register it is undecidable whether the variable must be live in the return instruction on line 5 in Figure 8. A register must be assigned to `%x1`, as it is used on line 3, although whether

it must live across function calls can reduce the possible registers depending on the target platform and its calling conventions. Because of this, any sound approach to liveness analysis will be an approximation.

In some cases it is simply impossible to assign each variable its own register without conflict along interference edges, in which case one or more variables need to be assigned to memory instead. This is also referred to as *spilling* the variable/register to the stack, or the variable itself is referred to as *spilled*.

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. This forces every variable to be assigned a different register, which is a viable approach for small programs with fewer variables than the set of assignable registers. Such a heuristic is greedy in the sense that it picks an assignment known to be safe for the least amount of preprocessing possible. However, this can be a very inefficient assignment at runtime. When the number of variables is greater than the number of assignable registers, a variable must be spilled. A number of n variables greater than k working registers causes $n - k$ variables to be spilled, which can greatly reduce performance, but allows for translation in linear time.

4.1 Dataflow Analysis

Another approach, which is a more precise approximation, is a specific variant of the dataflow analysis as described in *Modern Compiler Implementation in ML* [3] and *Compilers: Principles, Techniques, and Tools* [4]. In general, dataflow analysis is the process of finding the possible paths in which data may propagate through various 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* [4, p. 599] 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:

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

with the $use[n]$ set being defined as any variables that may be used and the $def[n]$ as the set of variables defined in node n. The $def[n]$ either consists of one variable or equal to $\emptyset$. The $use[n]$ set is effectively unbounded as some instructions take any m variables as parameters.

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* [4, p. 605], whose operator depends on the type of dataflow analysis as well, but for liveness analysis a union is performed on the *live-in* sets of any successive nodes:

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

Initially, two empty 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 two equations above are applied iteratively until a fixed point is reached, i.e. an invariant point where neither $in[n]$ or $out[n]$ is changed for all instructions n.

The simplest equations to implement are the `def` and `use` functions, as all of the values of interest are located immediately within the instruction itself and not hidden behind some layer of indirection:

```
1 let def (s : S.SS.t) (insn : Cfg.insn) =
2   match insn with Insn (Some dop, _) -> S.SS.add dop s | _ -> s
```

```
let use (s : S.SS.t) (insn : Cfg.insn) =
  let op o s = match o with Ll.Id i -> S.SS.add i s | _ -> s in
  let po = Fun.flip op in
  match insn with
  | Insn (_, AllocaN (_, (_, o))) (* | Bitcast _ | ... | Zext _ *) ->
      op o s
  | Insn (_, Binop (_, _, l, r)) (* | Icmp _ | Store _ *) ->
      op l s |> op r
  | Insn (_, Call (_, _, args)) -> List.map snd args |> List.fold_left po s
  | Insn (_, Gep (_, bop, ops)) -> List.fold_left po (op bop s) ops
  | Insn (_, Select (c, (_, l), (_, r))) -> op c s |> op l |> op r
  | Insn (_, PhiNode (_, ops)) -> List.map fst ops |> List.fold_left po s
  | Term (Ret (_, Some o) | Cbr (o, _, _)) -> op o s
  | _ -> s
```

Where `S.SS` is a `Set.S` module built over the `Symbol.symbol` type found in `lib/symbol.ml`. The `flowin` function corresponds to the *live-in* equation (1) and is implemented as follows:

```
let flowin (i, insn) =
  let newin = S.SS.union (use insn) (S.SS.diff out.(i) (def insn)) in
  let changed = not (S.SS.equal newin in_.(i)) in
  if changed then in_.(i) <- newin;
  changed
```

And the `flowout` function which corresponds to the *live-out* equation (2) implemented as follows:

```
let flowout (i, _) =
  let newout =
    let succ = Cfg.G.succ g ids.(i) in
    List.fold_left
      (fun s v -> S.SS.union s in_.(Cfg.G.V.label v))
      S.SS.empty succ
  in
  let changed = not (S.SS.equal newout out.(i)) in
  if changed then out.(i) <- newout;
  changed
```

The set operations are applied as you would with the `S.SS.t` struct, the only interesting part is noting whether changes were made, as this is needed further down in execution. The actual dataflow analysis performed recursively as follows:

```
let dataflow (insns : Cfg.insn list) (ids : Cfg.G.V.t array) (g : Cfg.G.t) =
  let insns = List.mapi (fun i v -> (i, v)) insns |> List.rev in
  let in_ = Array.init (List.length insns) (fun _ -> S.SS.empty) in
  let out = Array.init (List.length insns) (fun _ -> S.SS.empty) in
  let rec dataflow () =
    let flowout = (* ... *)
    let flowin = (* ... *)
    let flow changed insn = changed || flowout insn || flowin insn in
    if List.fold_left flow false insns then dataflow () else (in_, out)
  in dataflow ()
```

Figure 9: Overall implementation of `dataflow`

One particularly noteworthy thing is reversing of the flattened list of instructions on line 2 of Figure 9. After enumerating each entry so that the corresponding vertex can be looked up in the `ids` array, it is reversed. The motivation for this is the fact that the $out[n]$ of node n depends on $in[s]$ for all successors s. Since most nodes only have one immediate successor, populating this first is significantly more efficient.

Consider a simple program like Figure 10 that only prints integers incrementally up until `argc`, in LLVM-- that would consist of one loop with one phi node as seen in Figure 11.

```c
#include <stdio.h>

int main(int argc, char **argv) {

  for (int i = 1; i <= argc; i++) {
    printf("%d!\n", i);
  }

  return 0;
}
```

Figure 10: `cat tests/count.ll`

```
define i32 @main (i32 %0, i8* %1) {
 %3 = icmp slt i32 %0, 1
 br i1 %3, label %4, label %5
4:
 ret i32 0
5:
 %6 = phi i32 [%8, %5], [1, %2]
 %7 = call i32 (i8*, ...) @printf (i8* @.str, i32
 %8 = add i32 %6, 1
 %9 = icmp eq i32 %6, %0
 br i1 %9, label %4, label %5
}
```

Figure 11: Stripped `clang -O1 -S -emit-llvm`

If one were to note the $in[s]$ and $out[s]$ sets for each instruction s for each iteration, one would arrive at Tables 1 and 2. The number of iterations is nearly halfed when reverseing the list of instructions prior to dataflow analysis as can be seen on how many columns are present in each.

i	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	out
0	0	3	0	3	0	3,8	0,8	3,8	0,8	3,8	0,8	0,3,8	0,8	0,3,8	0,8	0,3,8
1	3		3	8	3,8	8	3,8	8	3,8	0,8	0,3,8	0,8	0,3,8	0,8	0,3,8	0,8
2																
3																
4				8	8	8	8	8	8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
5	8	6	8	6	8	6	8	0,6	0,8	0,6	0,8	0,6	0,8	0,6	0,8	0,6
6	6	7	6	6	6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6
7	6	8	6	0,6	0,6	0,6	0,6	0,6	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8
8	0,6	9	0,6	9	0,6	9	0,6	8,9	0,6,8	8,9	0,6,8	8,9	0,6,8	0,8,9	0,6,8	0,8,9
9	9		9		9	8	8,9	8	8,9	8	8,9	0,8	0,8,9	0,8	0,8,9	0,8

Table 1: Output of `dune exec build -- -t lva -v tests/count1.ll`

i	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	out	in	out	in	out	in	out
0	0	3	0		0	3	0	3	0	3,8	0,8	3,8	0,8	3,8	0,8	3,8	0,8	3,8	0,8	3,8	0,8	0,3,8	0,8	0,3,8
1	3		3		3	8	3,8	8	3,8	8	3,8	8	3,8	8	3,8	8	3,8	0,8	0,3,8	0,8	0,3,8	0,8	0,3,8	0,8
2																								
3																								
4						8	8	8	8	8	8	8	8	8	8	8	8	0,8	0,8	0,8	0,8	0,8	0,8	0,8
5	8	6	8		8	6	8	6	8	6	8	6	8	0,6	0,8	0,6	0,8	0,6	0,8	0,6	0,8	0,6	0,8	0,6
6	6	7	6		6	6	6	6	6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6
7	6	8	6		6	0,6	0,6	0,6	0,6	0,6	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8	0,6	0,6,8
8	0,6	9	0,6		0,6	9	0,6	8,9	0,6,8	8,9	0,6,8	8,9	0,6,8	8,9	0,6,8	8,9	0,6,8	8,9	0,6,8	0,8,9	0,6,8	0,8,9	0,6,8	0,8,9
9	9		9	8	8,9	8	8,9	8	8,9	8	8,9	8	8,9	8	8,9	0,8	0,8,9	0,8	0,8,9	0,8	0,8,9	0,8	0,8,9	0,8

Table 2: Output of `dune exec build -- -t lva -v -r tests/count1.ll`

This is unnoticeable for snippets of marginal size like most examples included, but when applied to functions with a significant amount of def-use paths, it can be quite consequential. For `tests/sha256.ll` for instance, the running time of this analysis is effectively doubled (see Figure 12).

```
$ time dune exec build -- tests/sha256.ll -t lva -r
36.91s user 0,04s system 99% cpu 33,765 total
$ time dune exec build -- tests/sha256.ll -t lva
18,61s user 0,03s system 99% cpu 19,055 total
```

Figure 12: Average runtime over several runs (n=32)

4.2 Building an Interference Graph

The purpose of conducting dataflow analysis as above is finding variables that may be assigned the same register. This is done by building an interference graph, which is an undirected graph, whose nodes represent variables and edges that signify interference between them, i.e. variables a and b live at intersecting program points is represented with an edge (a, b).

The construction of an interference graph only depends on the *live-out* set and type of instruction. If the instruction defines a variable, said variable is in interference with all variables in the *live-out* set. There is one exception however: according to the Appel text, move instructions (i.e. phi nodes in the case of SSA form) are given special consideration. The purpose of phi nodes is to copy/move a certain value from a certain predecessor, so they are not necessarily in conflict for being live at the same time. Rather it would often benefit if they were assigned the same register to spare unnecessary moves.

Because of this, for any phi node of the form

$$a = \Phi(b_1, ..., b_n) \tag{3}$$

add edges to all *live-out* variables not in $\{b_1, ..., b_n\}$

$$\forall b_j \in out[i] \setminus \{b_1, ..., b_n\}, \text{add\_edge}(a, b_j),$$

and for any other instruction that defines a variable a

$$\forall b_j \in \{b_1, ..., b_n\}, \text{add\_edge}(a, b_j).$$

Although the Appel text [3] describes interference of variables with concrete registers as well as overlapping variables, this isn't considered in this implementation for simplicity's sake.

Again the `OCamlgraph` library is used, but this time `Imperative.Graph.Abstract` is parameterized over the aforementioned set type `S.SS.t`. Such that several variables can be assigned to the same node (more on this in section 5.2). A helper function is defined, that looks up a variable's symbol in the symbol table over vertices and adds it if it doesn't exist. Otherwise it simply returns the existing vertex identifier:

```
let vert g e t =
  match S.ST.find_opt e t with
  | Some v -> (v, t)
  | None ->
      let v = S.SS.add e S.SS.empty |> G.V.create in
      G.add_vertex g v;
      (v, S.ST.add e v t)
```

Then the interference graph is derived in Figure 13. First, all parameters are connected, then the instructions are processed as described in the Appel text [3, p. 222]: phi nodes, as they are the only `mov` instructions, interference edges are added with all $out[n] \setminus \{b_1, ..., b_n\}$ (see (3)) while all others are just $out[n]$.

```
let interf (params : Ll.uid list) (insns : Cfg.insn list) _ (out : S.SS.t array) =
  let g = G.create () in
  let vert2 t e = vert g e t |> snd and vert3 e t = vert g e t |> snd in
  let edge s1 s2 t =
    let v1, t = vert g s1 t in
    let v2, t = vert g s2 t in
    if v1 <> v2 then G.add_edge g v1 v2;
    t
  in
  let t = List.fold_left vert2 S.ST.empty params in
  let param t p1 =
    let param t p2 = if p1 <> p2 then edge p1 p2 t else t in
    List.fold_left param t params
  in
  let t = List.fold_left param t params in
  let setverts t s = S.SS.fold vert3 s t in
  let t = List.map (def S.SS.empty) insns |> List.fold_left setverts t in
  let ids = function Ll.Id id -> Some id | _ -> None in
  let defoutedges t (i, n) =
    let defs = def S.SS.empty n in
    let out =
      match n with
      | Cfg.Insn (_, Ll.PhiNode (_, ops)) ->
          List.map fst ops |> List.filter_map ids |> S.SS.of_list
          |> S.SS.diff out.(i)
      | _ -> out.(i)
    in
    let outedge e1 t = S.SS.fold (edge e1) out t in
    S.SS.fold outedge defs t
  in
  let t = List.mapi (fun i n -> (i, n)) insns |> List.fold_left defoutedges t in
  (t, g)
```

Figure 13: Overall implementation of `interf`

For instance, the LLVM-- program of Figure 14, a simple handwritten program that counts down from `%argc`, would translate the interference graph of Figure 15.

```
define i32 @main (i32 %argc, i8** %argv) {
entry:
 br label %loop
loop:
 %c1 = phi i32 [%argc, %entry], [%c2, %loop]
 call void @printf (i8* @fmt, i32 %c1)
 %c2 = sub i32 %c1, 1
 %cn = icmp sgt i32 %c2, 0
 br i1 %cn, label %loop, label %exit
exit:
 ret i32 0
}
```

Figure 14: `cat tests/countdown.ll`

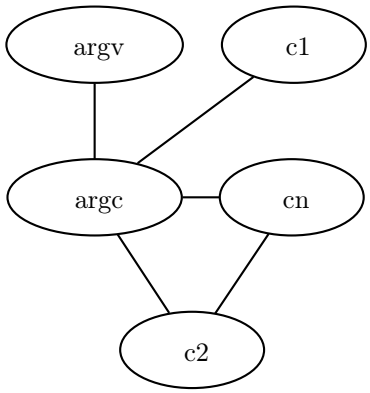


Figure 15: `dune exec build -- -t dot`

5 Graph Coloring

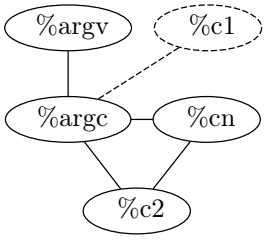
Once an interference graph is constructed, the actual assignments can be found using graph coloring. Graph coloring is a problem in graph theory where the goal is to assign colors to the vertices of a graph in such a way that no adjacent vertices are assigned the same color. A graph that can be colored using k different colors is referred to as k-colorable and the minimum number of colors needed to be assigned is also called its' chromatic number. Although this problem is known to be NP-complete [3, p. 229], [4, p. 510], the heuristic as introduced in both the Appel [3, p. 229], and Aho et al. [4, p. 557] texts, is a linear time approximation to this problem as described below.

5.1 Coloring by simplification

The *simplification* algorithm, also known as Chaitin's algorithm [16], is an iterative approach wherein nodes known to be colorable are removed and pushed to a stack until either an empty graph remains, in which case the original graph is k-colorable, or nodes with more than k neighbours remain. In this case a node is chosen to be spilled after which the process is started over. This is repeated until every node has been removed successfully, after which the vertices are assigned by popping them from the stack. Each of these are known to be k-colorable because of the earlier criteria of having less than k neighbors.

To perform the simplication algorithm on the previous example of Figure 14, its interference graph seen on Figure 15 only has one node of degree 4 with the ones remaining all being ≤ 2. This wouldn't be a problem on modern day processors most of which have ~16 or so GPRs, but for the sake of clearness one can imagine a register machine of $k = 2$ working registers.

If at first a single node of insignificant degree (i.e. < 2) is selected, it is then removed and pushed to a stack. In Figure 15 only `%argv` and `%c1` are of insignificant degree, so either of those is chosen to be removed (represented with dashed borders and edges) as can be seen on Figure 16a and 17a and pushed to the stack of variables to be colored as can be seen on Figure 16b and 17b.

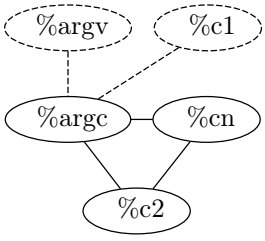


(a) Graph

`%c1`

(b) Stack

Figure 16: Push `%c1`

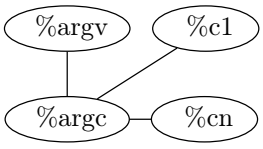


(a) Graph

`%argv`
`%c1`

(b) Stack

Figure 17: Push `%argv`

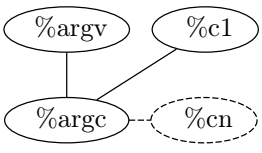


(a) Graph

(b) Stack

Figure 18: Spill `%c2`

From this point on nothing can be done to further simplify the graph as all of the remaining nodes are of significant degree (≥ 2). Therefore, one of the remaining nodes is spilled to memory and entirely removed from the graph as can be seen on Figure 18a and the stack reset as can be seen on Figure 18b.

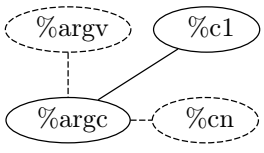


(a) Graph

`%cn`

(b) Stack

Figure 19: Push `%cn`

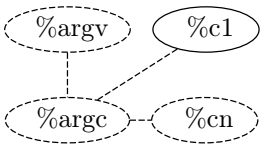


(a) Graph

`%argv`
`%cn`

(b) Stack

Figure 20: Push `%argv`



(a) Graph

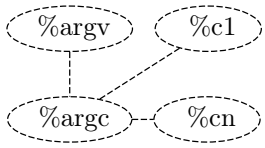
`%argc`
`%argv`
`%cn`

(b) Stack

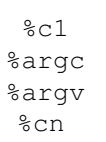
Figure 21: Push `%argc`

After spilling `%c2`, the process is repeated. First by pushing `%cn` (Figure 19a and 19b), then `%argv` (Figure 20a and 20b) and finally `%argc` (Figure 21a and 21b). One thing of note is the fact that the order in which

this is happening is irrelevant, so long as variables pushed are of insignificant degree. For instance, `%argc` was of degree 3 in Figure 18a, but as more and more of its neighbors are pushed its degree decreases to a point where only `%argc` and `%c1` are yet to be pushed to the stack.

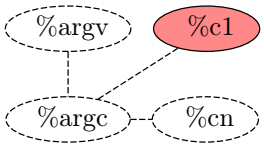


(a) Graph



(b) Stack

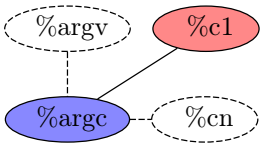
Figure 22: Push `%c1`



(a) Graph

(b) Stack

Figure 23: Pop `%c1`

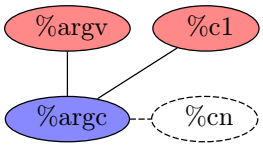


(a) Graph

(b) Stack

Figure 24: Pop `%argc`

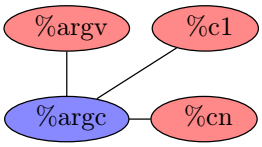
As the last variable `%c1` is pushed in Figure 22 and an empty graph remains, the process of reconstructing the graph can begin. When popping `%c1` in Figure 23, it can be assigned any of the two colors as it has no neigbors adjacent, so either is chosen. On the other hand, when assigning `%argc`, it's only adjacent neighbor `%c1` is already assigned red so only blue remains. Likewise, for `%argv` and `%cn` their only immediately adjacent neighbor of `%argc` causes them to be assigned red as can be seen on Figure 25 and 26.



(a) Graph

(b) Stack

Figure 25: Pop `%argv`



(a) Graph

(b) Stack

Figure 26: Pop `%c1`

The final coloring of Figure 26 leads to the assignment

$$\textit{assign}[\texttt{\%argv}] = \textit{assign}[\texttt{\%c1}] = \textit{assign}[\texttt{\%cn}] = \texttt{\%red}$$
$$\textit{assign}[\texttt{\%argc}] = \texttt{\%blue}$$

and the spilled variable `%c2`

$$\textit{assign}[\texttt{\%c2}] = \texttt{stackslot(0)}$$

5.2 Coalescing

Coalescing is the process of eliminating moves/copies of data from one GPR to another by combining their interference graph nodes. This is similar to but not the same as the lack of interference edges between variables that are subject to move operations. This is because variables a and b may still be assigned different registers or even spilled if, for instance, either of them are of significant degree.

Coalescing joins nodes a and b to node ab preserving the edges of both to maintain soundness. Since all edges are preserved, the resulting node ab may be of a much higher degree. Because of this, only strategies that produce a k-colorable graph are worth considering, as additional spills negate the purpose entirely. One of the strategies introduced in the Appel text is the *Briggs* strategy [3, p. 232], in which nodes a and b can be coalesced iff the resulting node ab has $\leq K$ nieghbors of significant degree (i.e. $\geq K$).

Since the only move operations in use in LLVM-- are phi nodes, only variables that are subject to this sort of operation are eligible to be coalesced in this form. Variables that can be coalesced can be thought of as having a preference edge, and any nodes with such an edge stand to benefit from coalescing, but this may only be done if they are not in interference.

Consider the previous example (Figure 14): when applied with the same $k = 2$ nothing happens regardless of which allocator is used (see Figure 27 which was generated using the `briggs` allocator). This is because although the `%c1` and `%c2` nodes are not in interference, both neighbors of `%c2` are of significant degree

($deg(\texttt{\%cn}) = 2 \geq 2$ and $deg(\texttt{\%argc}) = 3 \geq 2$). Therefore they don't satisfy the Briggs criteria, and are not coalesced, resulting in the same interference graph as the `simple` allocator in Figure 15. But for some $k \geq 2$, now there are fewer than k significant neighbors of `%c2` (and `%c1`), meaning the two can be coalesced, with the resulting node `%c1, %c2` having two neighbors of insignificant degree as can be seen in Figure 28.

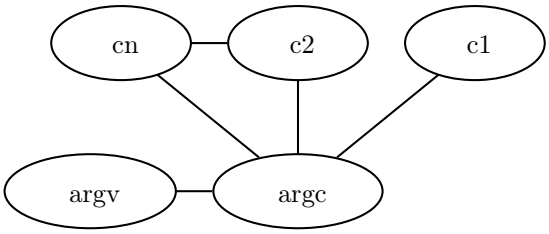


Figure 27: `build -- -t dot -a briggs -n 2`

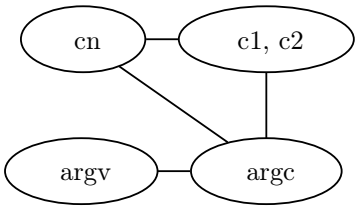


Figure 28: `-a briggs -n 3`

The purpose is only to assign the same registers to different variables, which stands to benefit from less `mov` instructions being used.

5.3 Implementation

The first step is deriving all preference edges:

```
let prefer (insns : Cfg.insn list) : S.SS.t S.ST.t =
  let insn t = function
    | Cfg.Insn (Some d, Ll.PhiNode (_, ops)) ->
        List.fold_left
          (fun t o ->
            match o with
            | Ll.Id sop, _ ->
                S.ST.update sop
                  (function
                    | Some s -> Some (S.SS.add d s)
                    | None -> Some (S.SS.singleton d))
                  t
            | _ -> t)
          t ops
    | _ -> t
  in
  List.fold_left insn S.ST.empty insns
```

Figure 29: Implementation of `prefer` in `lib/coalesce.ml`

This is done simply by iterating over all of the instructions, and for every phi node, add a preference edge between a on the left-hand side of (3) and all b_i on the right. Of note is the fact that only operands of the form $\%i$ (i.e. `Id` variants of the `operand` type) are considered. This is because immediate values may also be specified in an LLVM phi node, whose `mov` instruction don't make sense to eliminate.

Briggs coalescing is then implemented as follows:

```
let coalesce_briggs k (prefs : S.SS.t S.ST.t)
    ((l, g) : Lva.G.V.t S.ST.t * Lva.G.t) : Lva.G.V.t S.table * Lva.G.t =
  let try_coalesce sop dop (l, g) =
    if not (Lva.G.mem_edge g (S.ST.find sop l) (S.ST.find dop l)) then
      let sneighs = S.ST.find sop l |> Lva.G.succ g |> Lva.VS.of_list
      and dneighs = S.ST.find dop l |> Lva.G.succ g |> Lva.VS.of_list in
      let neighs = Lva.VS.union sneighs dneighs in
      let sign v = Lva.G.succ g v |> List.length >= k in
      let signeighs = Lva.VS.filter sign neighs in
      if Lva.VS.cardinal signeighs < k then
        coalesce (S.ST.find sop l) (S.ST.find dop l) (l, g)
      else (l, g)
    else (l, g)
  in
  let try_pref sop dops (l, g) = S.SS.fold (try_coalesce sop) dops (l, g) in
  S.ST.fold try_pref prefs (l, g)
```

Figure 30: Implementation of `coalesce_briggs` in `lib/coalesce.ml`

For every preference edge, attempt to coalesce them into one. The `try_coalesce` function is applied to every pair (a, b) for all Φ-nodes. At first whether a and b are in conflict is checked, because only nodes that are not in interference may be coalesced, and if they are there is no point in finding their neighbors. The degree of potential node ab is taking all of the neighbors of a and b, and if the union of these is of magnitude $\geq k$ it doesn't satisfy the Briggs criteria and no further action is taken. If it is $< k$, i.e. node ab is of insignificant degree, the two are combined in the `coalesce` function.

The following is a very tedious implementation, stemming from the the `ocamlgraph` library simply doesn't support contracting a graph. Instead one is directed towards constructing a new graph and copying over the vertices and edges in that order. This is done by first folding over all existing vertices, adding the one's that are not in the process of being coalesced, and when the first of the pair being coalesced is met, set the third tuple entry to be some, the associated data being the vertex identifier as well as it's associated symbols. Each vertex maps to a symbol set (i.e. `S.SS.t`) of variables already coalesced which is initially a singleton.

```
let coalesce v1 v2 (st, g) =
  if v1 = v2 then (st, g)
  else
    let g' = Lva.G.create () in
    let st', vt', _ =
      Lva.G.fold_vertex
        (fun v (st, vt, jn) ->
          if v = v1 || v = v2 then (
            match jn with
            | None ->
                let s = Lva.G.V.label v in
                (st, vt, Some (v, s))
            | Some (v1, s1) ->
                let s' = Lva.G.V.label v |> S.SS.union s1 in
                let v' = Lva.G.V.create s' in
                Lva.G.add_vertex g' v';
                ( S.SS.fold (fun e l -> S.ST.add e v' l) s' st,
                  Lva.VT.add v1 v' vt |> Lva.VT.add v v',
                  None ))
          else
            let s = Lva.G.V.label v in
            let v' = Lva.G.V.create s in
            Lva.G.add_vertex g' v';
            ( S.SS.fold (fun e st -> S.ST.add e v' st) s st,
              Lva.VT.add v v' vt,
              jn ))
        g
        (S.ST.empty, Lva.VT.empty, None)
    in
    Lva.G.iter_edges
      (fun v1 v2 ->
        let v1 = Lva.VT.find v1 vt' in
        let v2 = Lva.VT.find v2 vt' in
        if v1 <> v2 then Lva.G.add_edge g' v1 v2)
      g;
    (st', g')
```

Figure 31: Implementation of `coalesce` in `lib/coalesce.ml`

6 Linear Scan

Another approach that depends on liveness analysis is the linear scan algorithm. It is a simple and efficient approach that performs a linear pass over IR to produce a register assignment significantly faster than the steps involved with graph coloring. That said, it may generate more imprecise assignments and therefore also be less performant at runtime, so the utility of such an approach depends on the context in which it is used. While compilers like clang translate to machine code ahead-of-time (AOT), languages like C# and Java will often rely on just-in-time (JIT) compilation to translate their respective VM instructions to native machine code. Such VMs that depend on JIT will often defer compilation to runtime. This is done for a myriad of reasons, portability and security among them, but usually at the cost of runtime performance. One of the reasons for this can be attributed to the way in which registers are allocated, as a significant downside of JIT is the fact it needs to be compiled in full before execution can begin. Therefore, generally a faster class of allocators is employed, linear scan being one of them, as linear scan is a reasonably performant alternative

to graph coloring, that may be used to achieve reasonably efficient register assignments at runtime.

The primary text is *Linear Scan Register Allocation* [5], the application of which also relies on prior liveness analysis. Given liveranges, assignments are easily derived from only one pass through IR. Similar to graph coloring, given k available GPRs, it assigns as many variables as possible, and spills the rest.

It does this by first building the intervals using the $in[s]$ and $out[s]$ resulting from prior dataflow analysis. By keeping a list of active variables, on reaching the beginning of a new interval whith lk variables already actuve, one of them need to be spilled such that the starting interval can be assigned a register. It could be chosen at random but that could be costly, instead some spilling heuristic is applied.

6.1 Spilling heuristics

The spilling heuristic can vary, but the one mentioned in the paper is chosen from remaining length, as in *'furthest away from the current point'* [5, p. 900]. The same decision was made for this implementation, as it does stand to reduce the amount of spills necessary. See The program of Figure 33 and its associated live ranges/intervals of Figure 32. For a target of $k = 2$, upon reaching the start of interval `%5`, one variable needs to be spilled. If either `%3` or `%4` are chosen, in the case of `%3` being spilled, already upon reaching `%7` another needs to be spilled. In case of `%4`, two variables won't be live simultaneously until `%8`. Nonetheless, that means two spills instead of spilling the longest interval of `%5`.

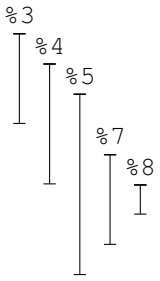


Figure 32: Live intervals

```
define i32 @main (i32 %0, i8* %1) {
  %3 = getelementptr i8*, i8** %1, i64 1
  %4 = load i8*, i8** %3
  %5 = call i32 @atoi (i8* %4)
  call i32 @printf (i8* @.str, i8** %3, i8* %4, ...
  %7 = call i32 @isqrt (i32 %5)
  %8 = call i64 @strtoll (i8* %4, i8* null, i32 10)
  call i32 @printf (i8* @.str.1, i32 %5, i64 %8)
  call i32 @printf (i8* @.str.2, i32 %7, i32 %5)
  ret i32 %5
}
```

Figure 33: `@main` of `tests/linear.ll`

6.2 Implementation

At first, intervals are derived from the flattened instruction sequence and associated $in[n]$ and $out[n]$ sets. While this can be done in a single pass with some creative array indexing, but my approach relies on two passes: one for finding the start points (see Figure 34), the other for finding the end points.

```
let add e = function
  | Some s -> Some (S.SS.add e s)
  | None -> Some (S.SS.singleton e)
let intervalstart insns (in_, out) =
  let insn (ordstarts, starts, active) (i, _n) =
    let out' = S.SS.diff (S.SS.union in_.(i) out.(i)) active in
    ( S.SS.fold (fun e a -> IT.update i (add e) a) out' ordstarts,
      S.SS.fold (fun e a -> S.ST.add e i a) out' starts,
      S.SS.union active out' )
  in
  List.fold_left insn (IT.empty, S.ST.empty, S.SS.empty) insns
```

Figure 34: Implementation of `intervalstart` in `lib/linear.ml`

Of note are the structures over which the folding is done: the `IT.t` is a `Map.S` module indexing earlier described symbol sets (i.e. `S.SS.t`). The reason for doing this is to iterate through intervals in increasing and decreasing order of start/length and end points respectively. The symbol table (`S.ST.t`) is to allow for reverse lookup, i.e. which program point variable s is defined, and lastly the symbol set (i.e. `S.SS.t`) which is just there to prevent multiple start points for the same variable.

```
let intervalends insns (in_, out) =
  let insn (i, _n) (ordends, ends, active) =
    let in' = S.SS.diff (S.SS.union in_.(i) out.(i)) active in
    ( S.SS.fold (fun e a -> IT.update i (add e) a) in' ordends,
      S.SS.fold (fun e a -> S.ST.add e i a) in' ends,
      S.SS.union active in' )
  in
  List.fold_right insn insns (IT.empty, S.ST.empty, S.SS.empty)
```

Figure 35: Implementation of `intervalendsintervalstart` in `lib/linear.ml`

The opposite is quite similar (see Figure 35), with the only real difference being the use of `List.fold_right` instead of `List.fold_left` to reduce the function body in reverse.

```
let intervals insns (in_, out) =
  let insn (starts, t, active) (i, n) =
    let init s (added, t, active) = (* .. *) in
    let end_ s (added, t, active) = (* .. *) in
    let a, t, c =
      init in_.(i) (S.SS.empty, t, active) |> init out.(i) |> end_ out.(i)
    in
    ((i, n, a) :: starts, t, c)
  in
  let starts, _, active =
    List.fold_left insn
      ([], S.ST.empty, Array.init (List.length insns) (Fun.const S.SS.empty))
      insns
  in
  List.rev starts |> List.map (fun (i, n, a) -> (i, n, a, active.(i)))
```

Figure 36: Implementation of `intervals` in `linear.ml`

The array indexing is used in the inner functions of `init` and `end_` to unset the last point at which an interval goes out of scope and are defined in Figure 37 and 38 respectively.

```
let init s (added, t, active) =
  ( S.SS.filter (fun s -> not (S.ST.mem s t)) s |> S.SS.union added,
    S.SS.fold
      (fun s t ->
        S.ST.update s
          (function
            | Some j ->
                active.(i) <- S.SS.add s active.(i);
                active.(j) <- S.SS.remove s active.(j);
                Some i
            | None ->
                active.(i) <- S.SS.add s active.(i);
                Some i)
          t)
      s t,
    active )
```

Figure 37: Implementation of `init` in `linear.ml`

```
let end_ s (added, t, active) =
  ( S.SS.filter (fun s -> not (S.ST.mem s t)) s |> S.SS.union added,
    S.SS.fold
      (fun s t ->
        S.ST.update s
          (function
            | Some j ->
                active.(i) <- S.SS.add s active.(i);
                active.(j) <- S.SS.remove s active.(j);
                Some i
            | None -> Some i)
          t)
      s t,
    active )
```

Figure 38: Implementation of `end_` in `linear.ml`

```
let linearscan k insns =
  let avail = Regs.of_list (List.init k reg_of_int) in
  let insn (avail, active, assign) (_, _n, s, _e) =
    (*Printf.printf "%s\n" (Cfg.string_of_insn n);*)
    let start s (avail, active, assign) =
      match Regs.choose_opt avail with
      | Some reg -> (Regs.remove reg avail, S.ST.add s reg active, assign)
      | None ->
          (* FIXME: order by remaining or total length *)
          let z, reg = S.ST.choose active in
          ( avail,
            S.ST.remove z active |> S.ST.add s reg,
            S.ST.add z
              (Ind3 (Lit (Int64.of_int ((S.ST.cardinal assign * -8) - 8)), Rbp))
              assign )
    in
    S.SS.fold start s (avail, active, assign)
  in
  let _, active, assign =
    List.fold_left insn (avail, S.ST.empty, S.ST.empty) insns
  in
  S.ST.fold (fun k v acc -> S.ST.add k (Reg v) acc) active assign
```

Figure 39: Implementation of `linearscan` in `linear.ml`

7 Instruction Selection

7.1 LLVM-- instruction set

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 interrupts, floating point operations etc., and instead only strive to cover the basics of compilers. In additional some other instructions are also included, like `trunc` and `switch`, to widen the range of supported frontend languages with the clang compiler.

7.2 Translating LLVM-- to x86

Translating each IR instruction to x86 correctly is a matter of eliminating unintended side-effects. Each LLVM-- instruction generally only serves one specific function, 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 [3, p. 190]. This 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.

Another more concise 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.

To map each LLVM-- instruction to what amounts to the 'core' function in x86, consider Table 3. Instructions for which the size varies, the suffix is denoted with x instead. This is then replaced with the

appropriate size suffix depending on the size denoted in the LLVM-- instruction. For example, most binary operations move the left-hand operand into `%rax` and the right-hand operand into `%rcx`, after which the associated instruction from the table is performed. For x86, most operations are done in place, so for the most part `%rax` is modified and moved to the destination after.

LLVM--	x86	Registers affected
add	addx %src, %dst	{}
alloca	subq $n, %rsp	{%rsp}
allocan	subq $n, %rsp	{%rsp}
and	andx %src, %dst	{}
ashr	sarx %src, %dst	{}
bitcast	movx %src, %dst	{}
call	*see 7.2.1*	{%rax, %rcx, %rdx, %rsi, %r8–%r11}
gep	*see 7.2.2*	{}
icmp	*see 7.2.3*	{%rFLAGS}
load	movx (%src), %dst	{}
lshr	shrx %src, %dst	{}
mul	imulx %src, %dst	{%rax}
or	orx %src, %dst	{}
phi	*see 7.2.4*	{}
ptrtoint	movx %src, %dst	{}
sdiv	idivx %src, %dst	{%rax, %rdx}
select		{}
sext	movx %src, %dst	{}
shl	shl %src, %dst	{}
srem	idivx %src, %dst	{%rax, %rdx}
store	movx %src, (%dst)	{}
sub	subx %src, %dst	{}
trunc	movx %src, %dst	{}
udiv	divx %src, %dst	{%rax, %rdx}
urem	divx %src, %dst	{%rax, %rdx}
xor	xorx %src, %dst	{}
zext	movx %src, %dst	{}

Table 3: LLVM-- instruction to x86 translation table

7.2.1 Implementing `call` instructions

The **`call`** operation is by far the most complicated one. Mostly because of calling conventions, caller saved registers listed in Table 3 need to be saved/restored before/after all. This isn't necessary in all cases, only the cases where they are actually used (which ought to be a simple lookup in the assignments table), but is not accounted for in this implementation. Instead they are all saved/restored regardless of use this could also be further optimized by simply subtracting from `%rsp` and moving them to their respective stack locations manually, instead of using push/pop. Additionally, the `%rax` register is zeroed before every call as its lowermost byte register `%al` is used to indicate the number of variadic arguments [13, p. 25]. This isn't actually accounted for, instead it is just set to zero regardless. Furthermore, the way in which arguments are passed is also unnecessarily convoluted, but stems from a corner case in which variables were overwritten by moving to the respective register of the n function argument. This can be remedied by ordering the `movs` correctly, and even in the case where there are cycles and `movs` cannot be made without overwriting, one of the scratch registers can be used instead. But as a workaround for this, the arguments are simply pushed from the source operands and popped to their respective argument registers (in reverse order).

7.2.2 Implementing `getelementptr` instructions

GEP instructions are used to perform address calculations for accessing elements in structures or arrays. They are particularly useful for efficiently addressing elements in complex data structures. For example, code accessing fields like line 8 in Figure 40, would translate to the snippet seen in Figure 41.

```
1  typedef struct {
2    int x;
3    int y;
4  } vector;
5
6  void dot(vector *a, vector *b) {
7    a->x += b->x;
8    a->y += b->y;      // -------->
9  }
10
```

Figure 40: Subfield access

```
1  ; ...
2  %6 = getelementptr %struct.vector,
3               %i8* %1, i64 0, i32 1
4  %7 = load i32, i32* %6
5  %8 = getelementptr %struct.vector,
6               %i8* %0, i64 0, i32 1
7  %9 = load i32, i32* %8
8  %10 = add i32 %9, %7
9  store i32 %10, i32* %8
10 ; ...
```

Figure 41: `clang -O1 -S -emit-llvm`

7.2.3 Implementing `icmp` instructions

The **`icmp`** and **`select`** instructions both rely on the `cmpx` x86 instruction for most of the heavy lifting.

7.2.4 Implementing `phi` instructions

Another irregular instruction is the **`phi`** instruction representing the Φ-functions. Their intended purpose is to copy data from whichever predecessor node was executed. As opposed to the other instructions that are only concerned with transferring the current state to another, this operation needs to concern itself with what was executed immediately prior. Fortunately there is no need to further complicate the scenario with additional state for the past executed block, as simply moving from $assign[b_i]$ to $assign[a]$ immediately prior to branching into the block whose header contains a phi node of the form (3) will suffice.

```
1  define i32 @main (i32 %0) {
2  1:
3   %2 = icmp sgt i32 %0, 0
4   br i1 %2, label %3, label %9
5  3:
6   %4 = phi i32 [%0, %1], [%7, %3]
7   %5 = phi i32 [0, %1], [%6, %3]
8   %6 = add i32 %5, %0
9   %7 = sub i32 %4, 1
10  %8 = icmp ugt i32 %7, 0
11  br i1 %8, label %3, label %9
12 9:
13  %10 = phi i32 [0, %1], [%6, %3]
14  ret i32 %10
15 }
```

Figure 42: `cat tests/square0.ll`

```
1  _main$3:
2    # ...
3    # br i1 %8, label %3, label %9
4    movq %rdx, %rax
5    cmpq $0, %rax
6    movq %rbx, %rax
7    movq %rax, %rdx
8    je _main$9
9    movq %rsi, %rax
10   movq %rax, %rsi
11   movq %rbx, %rax
12   movq %rax, %rdx
13   jmp _main$3
14 _main$9:
15   # ...
```

Figure 43: `dune exec build -- -t x86`

Notice for instance Figure 42 which has three phi nodes: two in block 3 and one in block 9, i.e. `%0` needs to be copied to `%4` when control is transferred from block 1 and `%7` when transferred from block 3. Likewise the immediate value `$0` must be copied to `%5` when transferred from block 1 and `%6` when transferred from block

3. All of the relevant translation can be seen in the fragment on Figure 43. After the comparison operation on lines 4-6 that check if the boolean condition of `%2` is nonzero and setting `FLAGS` register accordingly, `$0` is moved into *assign*[`%1`] in preparation of transferring to `_main$9` if the zero flag indeed is set. If not, execution is continued in which case both *assign*[`%6`] and *assign*[`%7`] are moved into *assign*[`%4`] and *assign*[`%5`] respectively, in preperation for continuing execution at `_main$3`.

As `mov` operations do not change the `FLAGS` register [17], having `mov` instructions between the branching instructions of `je` and `jmp` do not affect the branching. Likewise, because of the *use*[*n*] semantics of Φ-functions, all b_i are in interference in one another, so may not be overwritten by the `mov`s inserted.

One particularly nasty oversight that was discovered a couple of days prior to handin stems from phi movs overwriting each other. Consider Figure 44 where two phi instructions either move `%6` to `label %10` or `%7` to `label %5` depending on the condition. In some cases, even in greedy where each variable is assigned their own discrete register, their respective values are overwritten. This is due to `%7` being moved to `%6` prior to potentially branching back to `label %5`. If phi moves are conducted unconditionally of branch condition `%9`, the value of `%6` prior to branching to `label %10` is overwritten on the off-chance of returning to `label %5`, causing the value assigned to `%11` and returned on line 10 to always be equal to `%7` instead of the intended `%7` instead of the intended `%6`.

```
1  ; ...
2  5:
3   %6 = phi i32 [%7, %5], [1, %3]
4   %7 = add i32 %6, 1
5   %8 = mul i32 %7, %7
6   %9 = icmp sgt i32 %8, %0
7   br i1 %9, label %10, label %5
8  10:
9   %11 = phi i32 [%0, %1], [0, %3], [%6, %5]
10  ret i32 %11
11 }
```

Figure 44: `dune exec build -- -t x86`

```
1  # ...
2  # br i1 %9, label %10, label %5
3  movq        %r10, %rax
4  cmpq        $0, %rax
5  movq        %r8, %rax
6  movq        %rax, %rdi
7  je        _isqrt$5
8  movq        %rdi, %rax
9  movq        %rax, %r11
10 jmp       _isqrt$10
11 # ...
```

Figure 45: TODO

This was resolved by relying on conditional `mov`s immediately prior to the conditional branch (i.e. `je`). Specifically, the `cmoveq` instruction was used, which only moves from source to destination if the equality flag (i.e. `ZF`) is set. Unfortunately the `cmoveq` instruction doesn't support indirect addressing, so in order to support spilled variables both the values need to be moved into registers as can be seen in Figure 47.

```
1  # ...
2  # br i1 %9, label %10, label %5
3  movq        %rdx, %rax
4  cmpq        $0, %rax
5  movq        %rbx, %rax
6  cmoveq        %rax, %rsi
7  je        _isqrt$5
8  movq        %rsi, %rax
9  movq        %rax, %rdx
10 jmp       _isqrt$10
11 # ...
```

Figure 46: TODO

```
1  # ...
2  # br i1 %13, label %14, label %9
3  movq        -88(%rbp), %rax
4  cmpq        $0, %rax
5  movq        -72(%rbp), %rax
6  movq        -16(%rbp), %rcx
7  cmoveq        %rax, %rcx
8  movq        %rcx, -16(%rbp)
9  je        _main$9
10 movq        -16(%rbp), %rax
11 # ...
```

Figure 47: `-a greedy -n 0`

7.3 Assessing Correctness

It is difficult to assert the correctness of a compiler. While writing simple unit tests is easy, guaranteeing the correctness of code generated for any given input is on a different order of magnitude in terms of complexity.

Validating the correctness of each instruction translation is a complicated task given how low-level it is. One approach is attaching a debugger and examining the state before and after the translated sequence of x86 instructions is executed. The result of applying a binary operation, for instance, should only affect the assigned register. Another is an extensive test suite, although achieving 100% coverage is unrealistic.

7.4 Test Suite and Framework

The test suite is quite extensive although not anywhere close to complete coverage. Some tests have been handwritten during development, mostly meant to assert one specific feature (unit tests), but others are also inteded to assert them working together (integration tests). Some of them are hand written (simple ones), others are based on the output of `clang -S -emit-llvm` stripped with the the strip utility in `bin/strip.ml`, and lastly some are generated with Python scripts to parameterize certain properties (e.g. long move-chains, many operations). In addition to this some are copied from other projects In addition to my own cases and other programs from the internet (attributed in the file header), some cases from the LLVM test suite are also used. The structure of the test suite repository has a directory of so-called 'single source' C programs totalling to 2113. Of these only 1222 compiled successfully with `clang -S -emit-llvm` and of these only 624 parsed sucessfuly with the strip utility, of which 425 compiled successfully. In addition to these there are also the test suite used throughout Tiger development, most of which were provided by the compilers lecturers, as well as some LLVM snippets from the Dolphin compiler.

The testing framework is roughly based on the build system that came with the Tiger development environment. It is based on a collection of files that can be compiled and executed on their own. Each of these are enumerated in the test runner (`bin/testrunner.ml`) with parameters associated. Each of these are then compiled with the `clang` compiler and executed, capturing all the relevant output, which is stdout and exit code. After this, the same test file is compiled for each of the allocators. The executables generated are then also executed, also storing the output data. All test cases are deterministic and will be halted by configurable timeout at some point. This is necessary because tests can easily end in an infinite loop because of erroneous translation. As each of these halt or time out, their output is compared to what was output from executables generated by the 'correct' compiler of `clang`. Though testing the outputs of several stages much like the Tiger compiler from the compilers course, this wasn't realized in time. All of the test entries in `bin/testrunner.ml` were either working at one point and the result of a regression or it was a work in progress that wasn't realized in time either. Except the $\widetilde{2}0\%$ of the LLVM test suite that were never working.

8 Evaluation

While there are many approaches to assessing the quality of translation, one of the primary means is to simply measure the time taken to execute the code generated. Assuming the translation is sound given the previous steps taken to validate correctness in translating LLVM-- to x86, it is a solid foundation for future optimizations. While there are certainly other factors worth measuring, only runtime performance is considered for this project.

8.1 Benchmarking

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.

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 processes running (Xorg etc.).

Additionally, all benchmarks are deterministic, meaning relevant values at all points across all runs are consistent. So while not completely equivalent due to address randomization, such noise should never leak to value space, and assuming values allocated are initialized before use, no remnants from other processes should leak to value space either. 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. Although realistically this will never happen in a lifetime, across n runs the run least affected by context switching will be the one with the least CPU cycles measured, 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 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 on Linux. The `perf` subcommand is used, as it starts a process and attaches from the very beginning, in addition to padding a `-e` flag specifying that only cycles are meant to be recorded and the `-x` flag to help parse the output. Both min, avg and max are tracked and reported but only min are represented in the matrices below.

Some of the `.ll` file present in the `benches` directory are listed here in a table denoting the input paramater on the left column and type of allocator used. The data measured is extracted from the `perf` as mentioned above. This is done in `bin/bench.ml` which also contains the commands and parameters to execute each benchmark.

8.1.1 `benches/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.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
8	139964	1.00041x	0.98897x	1.00518x	1.00397x	0.99405x	1.00683x	1.02014x
10	141471	1.00025x	1.00662x	1.00701x	1.01013x	1.01107x	0.99899x	1.03276x
12	143729	1.01286x	1.00840x	1.02403x	0.97714x	1.00222x	1.02376x	1.03188x
14	152717	1.00032x	1.03967x	1.03472x	1.03637x	1.02969x	1.03224x	1.17572x
16	173395	1.08069x	1.05265x	1.08045x	1.08271x	1.06793x	1.06925x	1.40948x
18	227988	1.15964x	1.15565x	1.16758x	1.16818x	1.16349x	1.16611x	1.80801x
20	375309	1.24200x	1.23989x	1.24662x	1.24412x	1.24000x	1.24497x	2.26177x
22	760328	1.30708x	1.31289x	1.31045x	1.31230x	1.30795x	1.30885x	2.63144x
24	1763965	1.35080x	1.35029x	1.35039x	1.35129x	1.35274x	1.35185x	2.83868x
26	4321515	1.38984x	1.39041x	1.38939x	1.39050x	1.39175x	1.39096x	2.98115x
28	11291147	1.37185x	1.37282x	1.37280x	1.37369x	1.37371x	1.37320x	2.96675x
30	28225827	1.42877x	1.42993x	1.42896x	1.43085x	1.43172x	1.43069x	3.09771x
32	73749658	1.43033x	1.42973x	1.42936x	1.42954x	1.43088x	1.42974x	3.10158x

Table 4: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

The measures in Table 4 are significantly equivalent to one another, with all allocators (except greedy 0) seeming to converge towards being 37% slower than Clang. This isn't particularly surprising, as the x86 translation is more concerned with preserving callee saved registers than necessary in pushing each and every one despite not being in use. Also, instead of pushing with the `push` instruction it would be faster to simply subtract from the `%rsp` register directly and using `mov` directly.

What's more interesting and related to the allocation is the fact that greedy 0 is more than twice as slow. This is presumably because the `fib` function that has 7 variables can assign each of them to a register for the first six allocators listed. However, the last one (greedy 0), has no assignable registers, so it must

necessarily start spilling from the very beginning. To elucidate this hypothesis, the same benchmark was performed but only for the greedy of decreasing k assignable registers as well as the Tiger compiler which has the same approach to spilling every variable prematurely and can be seen in Table 8:

arg(s)	clang	greedy 12	greedy 8	greedy 6	greedy 4	greedy 2	greedy 1	greedy 0	tiger
8	139813	0.99868x	1.00816x	1.00434x	1.00410x	1.01222x	1.00127x	1.01345x	1.01526x
10	141482	0.99391x	1.01001x	1.01231x	1.01469x	1.00529x	1.02271x	1.03397x	1.02248x
12	145091	1.00774x	0.99627x	1.01584x	1.01047x	1.01694x	1.07056x	1.06622x	1.04396x
14	149598	1.06186x	1.06462x	1.06184x	1.06665x	1.08504x	1.20101x	1.20679x	1.14215x
16	170028	1.09372x	1.09738x	1.12226x	1.13603x	1.15900x	1.42535x	1.43162x	1.29614x
18	230073	1.15757x	1.15498x	1.17517x	1.21433x	1.25770x	1.73753x	1.79427x	1.51243x
20	376308	1.24275x	1.24539x	1.28023x	1.35349x	1.41074x	2.22527x	2.26082x	1.81290x
22	762044	1.30899x	1.30974x	1.35873x	1.45410x	1.52442x	2.57640x	2.62467x	2.05193x
24	1765499	1.34932x	1.34929x	1.40559x	1.51369x	1.59160x	2.78143x	2.83690x	2.18519x
26	4399288	1.36595x	1.36628x	1.42456x	1.53853x	1.62035x	2.87046x	2.92749x	2.24179x
28	11290936	1.37293x	1.37379x	1.43301x	1.54853x	1.63198x	2.90889x	2.96589x	2.27035x
30	28716847	1.40466x	1.40503x	1.46656x	1.58655x	1.67357x	2.98560x	3.04538x	2.32981x
32	76582019	1.37696x	1.37647x	1.43704x	1.55462x	1.64074x	2.92847x	2.98745x	2.28358x

Table 5: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

While it's not a complete 1:1 in terms of runtime as the Tiger compiler seems to perform slightly better than Greedy 0 (by ~30% or so), that does make sense as there are marginally more instructions involved with each operation, not to mention the extra care taken to ensure calling conventions are upheld.

8.1.2 `benches/phis.ll`

Coalescing with the Briggs allocator is evidently quite beneficial, as the it does out perform most benchmarks. This is obviously because eliminating movs stand to gain. A more direct approach to demonstrating this is with parameterized benchmarks. While in theory a phi node has the potential to eliminate one `mov` instruction, in practice this might be more beneficial due to the implementation. The `benches/phis.py <n> <m>` script will generate a `.ll` program consisting of m phi nodes over n iterations.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 2	linear 12	greedy 12	greedy 0
	33699586	2.49352x	2.49358x	1.24889x	1.24890x	2.49352x	2.49355x	2.49351x	11.61953x
	33696431	2.54359x	2.55986x	1.49846x	1.49796x	2.49373x	2.49378x	2.54356x	11.52326x
	50475128	1.78694x	1.99722x	1.16621x	1.33250x	1.66480x	1.66483x	1.77143x	8.57567x
	67274018	1.74795x	1.87263x	1.04956x	1.06208x	1.49853x	1.24908x	1.52063x	6.57981x
	84029983	1.39938x	1.79869x	1.00002x	1.00003x	1.29955x	1.00004x	6.19846x	5.98194x
	100810530	1.49925x	1.83217x	0.99997x	0.99996x	1.24967x	0.99998x	4.60493x	5.80149x
	117587019	1.35669x	1.78907x	0.88585x	0.90488x	1.21403x	0.85730x	4.10513x	5.51438x
	134365885	1.49947x	1.81229x	0.87512x	0.87515x	1.18732x	0.87514x	3.63380x	4.81382x

1074877208

phis	clang	simple 12	simple 2	briggs 12	briggs 2	linear 2	linear 12	greedy 12	greedy 0
0	150762	0.99922x	1.00386x	1.00104x	0.99880x	1.00325x	0.99096x	1.00077x	0.99764x
1	263112	1.78765x	1.83810x	1.26448x	1.75223x	1.75782x	1.76278x	1.78716x	6.27904x
2	333724	1.44531x	1.58670x	1.09621x	1.19483x	1.39069x	1.39096x	1.45969x	5.28416x
3	399042	1.49027x	1.57311x	1.03136x	1.03867x	1.32618x	1.16354x	1.32695x	4.61992x
4	464640	1.28048x	1.56253x	1.00015x	0.99836x	1.21081x	0.99861x	3.96512x	4.48737x
5	529535	1.37214x	1.61569x	0.99931x	0.99959x	1.17952x	0.99948x	3.58774x	4.47259x
6	594553	1.27726x	1.61016x	0.91261x	0.92786x	1.16424x	0.89141x	3.38554x	4.46924x
7	659461	1.39945x	1.61499x	0.90111x	0.90219x	1.15128x	0.90144x	3.08855x	4.03288x

Table 6: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

phis	clang	simple 12	simple 2	briggs 12	briggs 2	linear 2	linear 12	greedy 12	greedy 0
	150756	1.00269x	1.00742x	1.00356x	1.00237x	0.99991x	1.00269x	0.99794x	1.00298x
	266509	1.76877x	1.81332x	1.24997x	1.74009x	1.73912x	1.74314x	1.76571x	6.19913x
	331770	1.49255x	1.59420x	1.10326x	1.20125x	1.39740x	1.39905x	1.48485x	5.31939x
	398255	1.49334x	1.57569x	1.03280x	1.04240x	1.32999x	1.16465x	1.33295x	4.62699x
	463887	1.28291x	1.55488x	1.00064x	1.00080x	1.21103x	0.99887x	3.97028x	4.49752x
	529682	1.37111x	1.61877x	0.99918x	1.00048x	1.18536x	0.99971x	3.58938x	4.44715x
	594307	1.27157x	1.60988x	0.91407x	0.92725x	1.16608x	0.89112x	3.38938x	4.46031x
	659352	1.39923x	1.61575x	0.90244x	0.90256x	1.15120x	0.90131x	3.08998x	4.03296x

Table 7: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 2	linear 12	greedy 12	greedy 0
	150790	0.99873x	1.00204x	1.00055x	1.00202x	1.00475x	0.99925x	0.99983x	1.00036x
	266858	1.76227x	1.81025x	1.24026x	1.73967x	1.73845x	1.73599x	1.76279x	6.18927x
	332932	1.47463x	1.58958x	1.09780x	1.19567x	1.39187x	1.39397x	1.45798x	5.30079x
	396964	1.49802x	1.58318x	1.03435x	1.04439x	1.32655x	1.16965x	1.33411x	4.64216x
	463143	1.27994x	1.56894x	1.00193x	1.00219x	1.21525x	1.00036x	3.97536x	4.50366x
	528926	1.37202x	1.62053x	1.00265x	1.00090x	1.18442x	1.00260x	3.59263x	4.46336x
	594806	1.27651x	1.60931x	0.91222x	0.92581x	1.16478x	0.89116x	3.38589x	4.46530x
	660398	1.39599x	1.60987x	0.89423x	0.90015x	1.14754x	0.90032x	3.08470x	4.02611x

Table 8: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.3 `benches/subset.ll`

A suitable example for nested loop iteration is `benches/subset.ll`. It's a very intuitive program that simply prints all 2^n of a set of n elements, given as command line arguments. This is done by incrementing i by 1 from 0 to $2^n - 1$ and then printing the jth entry of the arguments array iff the jth bit of i is set. This leads to 2^n repetitions of the outer loop and n of the inner, leading to $2^n n$ repetitions. Since

n	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
14	1.00000x	1.08770x	1.15188x	1.08832x	1.15813x	1.92564x	2.59504x	2.95798x
15	1.00000x	1.08304x	1.14485x	1.08536x	1.14973x	1.93885x	2.60300x	2.96001x
16	1.00000x	1.08622x	1.14309x	1.08345x	1.14388x	1.94770x	2.62809x	2.99463x
17	1.00000x	1.08533x	1.14335x	1.08652x	1.14605x	1.94922x	2.64844x	3.01363x

Table 9: Benchmark of `benches/sieven.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.4 `benches/factori32.ll`

Integer factorization is another interesting algorithm. Generally considered to be NP, here are the results of factorizing known primes of inreasing bit length. For $n = 1024$.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
16777213	167130	1.04160x	1.00651x	1.05411x	1.05372x	1.04686x	1.84213x	2.33215x	2.34511x
33554393	175409	1.06219x	1.06070x	1.06415x	1.06212x	1.06039x	2.12199x	2.78336x	2.79956x
67108859	185779	1.07588x	1.07923x	1.08373x	1.08043x	1.07895x	2.49571x	3.38089x	3.40121x
134217689	197302	1.12173x	1.10225x	1.12120x	1.12080x	1.11794x	3.00248x	4.17628x	4.20718x
268435399	218313	1.14571x	1.14655x	1.14531x	1.14376x	1.14572x	3.55232x	5.06141x	5.08687x
536870909	245682	1.18408x	1.18299x	1.18404x	1.18194x	1.18307x	4.20818x	6.06747x	6.13899x
1073741789	285913	1.21667x	1.21894x	1.21755x	1.21668x	1.21706x	4.89647x	7.20172x	7.23917x

Table 10: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.5 `benches/factori64.ll`

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
268435399	218996	1.07020x	1.10754x	1.07114x	1.10950x	1.09908x	1.61452x	2.38305x	2.38509x
536870909	247101	1.08257x	1.13159x	1.08546x	1.13164x	1.13364x	1.79438x	2.73111x	2.73643x
1073741789	285989	1.10503x	1.16197x	1.10359x	1.16233x	1.16392x	1.97079x	3.11293x	3.12110x
2147483647	341802	1.12053x	1.18807x	1.12297x	1.19256x	1.19163x	2.14407x	3.49973x	3.50847x
4294967291	419941	1.14339x	1.22187x	1.14366x	1.22078x	1.22093x	2.31858x	3.87528x	3.88825x
8589934583	530002	1.16154x	1.24776x	1.16318x	1.24673x	1.24866x	2.47972x	4.22355x	4.23487x
17179869143	685889	1.17835x	1.27310x	1.17878x	1.27365x	1.27361x	2.61753x	4.52244x	4.53682x
34359738337	905070	1.19204x	1.29498x	1.19218x	1.29535x	1.29450x	2.73632x	4.77726x	4.79190x

Table 11: Benchmark of `benches/fib.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.6 `benches/sieven.ll`

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
128	159130	1.00777x	1.00293x	1.00644x	1.01562x	1.27104x	1.29856x	1.29894x
256	163800	1.00726x	1.01200x	1.00852x	1.01440x	1.53225x	1.58048x	1.59431x
512	172096	1.01902x	1.02488x	1.01789x	1.01876x	2.03762x	2.13731x	2.15792x
1024	188591	1.03687x	1.03706x	1.02964x	1.03689x	2.93413x	3.09867x	3.15326x
2048	221875	1.04971x	1.06586x	1.04806x	1.06306x	4.34620x	4.64145x	4.72466x
4096	288132	1.08649x	1.09864x	1.08082x	1.10033x	6.25886x	6.69614x	6.84597x
8192	424179	1.10601x	1.12721x	1.10683x	1.12700x	8.26470x	8.85856x	9.07318x
16384	697378	1.12603x	1.15191x	1.12581x	1.15269x	10.00521x	10.73828x	10.98249x

Table 12: Benchmark of `benches/sieven.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.7 `benches/sha256.ll`

n	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
100	1.00000x	3.17253x	3.35873x	3.17637x	3.37014x	19.50758x	19.44391x	19.61921x
1000	1.00000x	4.54197x	4.83710x	4.54593x	4.83359x	31.35583x	31.26454x	31.56475x
10000	1.00000x	4.80821x	5.11478x	4.80648x	5.12019x	33.50659x	33.41428x	33.72138x

Table 13: Benchmark of `benches/sha256.ll` output by `dune exec bench -- -f fib -n 1000`

8.1.8 `benches/fannkuch-redux.ll`

arg	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
4	197975	0.93202x	1.08772x	0.94359x	1.37648x	1.16456x	1.09012x	1.14835x
5	193800	1.05122x	1.10079x	1.07640x	1.11231x	1.72450x	1.78685x	1.73329x
6	307184	1.28426x	1.32022x	1.25973x	1.33286x	3.81882x	3.85798x	3.90989x
7	1312573	1.50533x	1.56355x	1.51619x	1.53672x	6.40226x	6.44050x	6.32867x
8	11252155	1.60713x	1.63817x	1.52961x	1.61191x	7.25974x	7.25458x	7.28693x
9	115513112	1.57771x	1.62155x	1.55882x	2.57223x	7.60550x	7.87522x	7.69289x
10	1352747472	1.54700x	1.74875x	1.58381x	1.67440x	7.77269x	7.74449x	7.82170x

Table 14: Benchmark of `benches/sieven.ll` output by `dune exec bench -- -f fib -n 1000`

8.2 Comparison to other work and ideas for future work

9 Conclusion

References

[1] Y. Shi, K. Casey, M. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," eng, *ACM transactions on architecture and code optimization*, vol. 4, no. 4, pp. 1–36, 2008, ISSN: 1544-3566.

[2] J. Dean and P. Norvig. "Latency comparison numbers." (), [Online]. Available: https://web.archive.org/web/20240101182413/https://gist.github.com/jboner/2841832 (visited on 01/01/2024).

[3] A. W. Appel, *Modern Compiler Implementation in ML*, eng. Cambridge University Press, 1997, ISBN: 0521582741.

[4] A. V. Aho, M. S. lan, R. Sethi, and J. D. Ullmann, *Compilers : principles, techniques, and tools*, eng, 2nd ed., internat. ed. Harlow: Pearson Education Limited, 2014, ISBN: 1292024348.

[5] M. Poletto and V. Sarkar, "Linear scan register allocation," eng, *ACM transactions on programming languages and systems*, vol. 21, no. 5, pp. 895–913, 1999, ISSN: 0164-0925.

[6] L. Foundation. "Clang: A c language family frontend for llvm." (), [Online]. Available: https://web.archive.org/web/20231230181618/https://clang.llvm.org/ (visited on 12/30/2023).

[7] D. L. Foundation. "Ldc – the llvm-based d compiler." (), [Online]. Available: https://web.archive.org/web/20231230181935/https://github.com/ldc-developers/ldc (visited on 12/30/2023).

[8] A. Inc. "Ldc – the llvm-based d compiler." (), [Online]. Available: https://web.archive.org/web/20231230182842/https://developer.apple.com/swift/ (visited on 12/30/2023).

[9] A. Inc. "Ldc – the llvm-based d compiler." (), [Online]. Available: https://web.archive.org/web/20231230185321/https://rustc-dev-guide.rust-lang.org/overview.html (visited on 12/30/2023).

[10] L. Foundation. "Writing an llvm backend." (), [Online]. Available: https://web.archive.org/web/20231230200453/https://llvm.org/docs/WritingAnLLVMBackend.html (visited on 12/30/2023).

[11] . F. Martin Woodward Executive Director. "Announcing llilc - a new llvm-based compiler for .net." (2015), [Online]. Available: https://web.archive.org/web/20211212184833/https://dotnetfoundation.org/blog/2015/04/14/announcing-llilc-llvm-for-dotnet (visited on 09/12/2020).

[12] K. Foundation. "Kotlin native." (), [Online]. Available: https://web.archive.org/web/20231230192142/https://kotlinlang.org/docs/native-overview.html (visited on 12/30/2023).

[13] *System v application binary interface amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0*.

[14] "Llvm language reference manual - functions." (Jan. 13, 2024), [Online]. Available: https://releases.llvm.org/9.0.0/docs/LangRef.html#functions.

[15] "Ocamlgraph." (Jan. 2, 2024), [Online]. Available: https://github.com/backtracking/ocamlgraph.

[16] G. Chaitin, "Register allocation & spilling via graph coloring," eng, in *Proceedings of the 1982 SIGPLAN symposium on compiler construction*, ACM, 1982, pp. 98–105, ISBN: 9780897910743.

[17] "X86 opcode and instruction reference 1.12." (Jan. 4, 2024), [Online]. Available: https://web.archive.org/web/20240104142515/http://ref.x86asm.net/geek64-abc.html#M.