

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

William Welle Tange – 201809080  
Advisors: Aslan Askarov & Amin Timany

January 15, 2024



Bachelor report (15 ECTS) in Computer Science  
Department of Computer Science, Aarhus University

## Abstract

In this day and age, compilers are used in more places than one would immediately consider. While programmers heavily rely on them to get their work done in a reasonable amount of time, as the alternative of writing assembly by hand is a tremendously time consuming not to mention risky feat in terms of security, a significant amount of time that could otherwise have been spent being productive is spent waiting on the compiler to finish, hence the productivity in the field of software development stands to benefit from improved efficiency. Likewise, scripting languages that used to be run by interpreting source text chunks at a time are now being translated to raw machine instructions immediately prior to execution just to squeeze out fractions of performance benefits on computers. As Moore's law seems to be nearing its natural limit as dictated by the laws of our universe, even the faintest reduction in compile-time will stand to benefit the runtime performance of an entire class of programming languages. Therefore this paper will seek to implement and evaluate the approach most often used in *ahead-of-time* compilation seeking to benefit the former, as well as the approach most often used in cases of *just-in-time* compilation for the latter.

## Acknowledgments

I would like to express my sincere gratitude and thanks to all who have supported me in making this project a part of reality. First of all, I can hardly begin to express my gratitude towards my advisors Aslan and Amin for the weekly meetings where most if not all concerns were settled. I'm also grateful for any fellow students who gave me inspiration and some insight along the way, naturally including some additional test cases that did manage to eek out some bugs. Additionally, all of the program snippets I have found online around the place have been quite helpful and saved me a lot of time, including of course the source of some benchmarks. Lastly I appreciate the lengths authors of the source material used in this paper has gone to to further the field of computer science and knowledge of mankind.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Intermediate Representation</b>	<b>1</b>
2.1	Static Single-Assignment form . . . . .	2
<b>3</b>	<b>Control Flow Analysis</b>	<b>3</b>
3.1	Building a Control Flow Graph . . . . .	4
<b>4</b>	<b>Liveness Analysis</b>	<b>5</b>
4.1	Dataflow Analysis . . . . .	6
4.2	Building an Interference Graph . . . . .	9
<b>5</b>	<b>Graph Coloring</b>	<b>11</b>
5.1	Coloring by simplification . . . . .	11
5.2	Coalescing . . . . .	12
5.3	Implementation . . . . .	13
<b>6</b>	<b>Linear Scan</b>	<b>15</b>
6.1	Spilling heuristics . . . . .	16
6.2	Implementation . . . . .	16
<b>7</b>	<b>Instruction Selection</b>	<b>19</b>
7.1	LLVM-- instruction set . . . . .	19
7.2	Translating LLVM-- to x86 . . . . .	19
7.2.1	Translating <code>call</code> instructions . . . . .	20
7.2.2	Translating <code>getelementptr</code> instructions . . . . .	21
7.2.3	Translating <code>phi</code> instructions . . . . .	22
7.2.4	Translating terminators . . . . .	23
7.3	Assessing Correctness . . . . .	23
7.4	Test suite/framework . . . . .	24
7.4.1	Debug Harness . . . . .	24
<b>8</b>	<b>Evaluation</b>	<b>24</b>
8.1	Benchmarking . . . . .	25
8.1.1	<code>benches/fib.ll</code> . . . . .	25
8.1.2	<code>benches/loopn.ll</code> . . . . .	26
8.1.3	<code>benches/phis.ll</code> . . . . .	28
8.1.4	<code>benches/factori32.ll</code> . . . . .	28
8.1.5	<code>benches/sieven.ll</code> . . . . .	28
8.1.6	<code>benches/sha256.ll</code> . . . . .	29
8.2	Future work . . . . .	29
<b>9</b>	<b>Conclusion</b>	<b>30</b>

# 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).

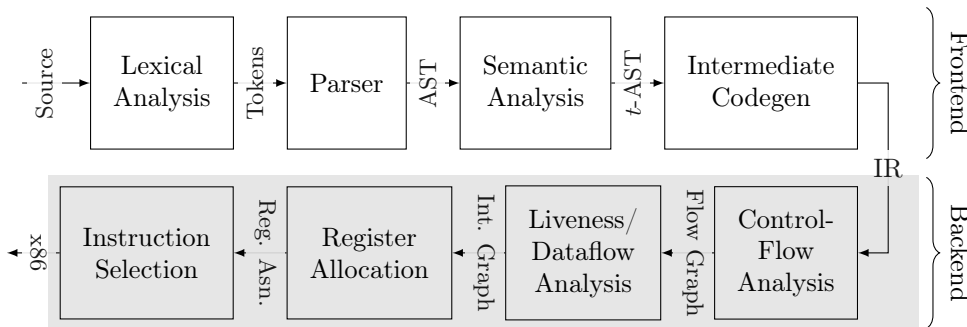


Figure 1: Compiler phases, backend highlighted

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 parameterized over some set of value as well as an attempt to further assert the correctness of translation.

## 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/debug 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.

```
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:
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 strictly 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 has 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.

```
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  $n$ th 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.

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

```

3 let verts = Array.init (List.length insns) G.V.create in
4 let addbl t = function i, Label l -> S.ST.add l verts.(i + 1) t | _ -> t in
5 let blocks = List.fold_left addbl S.ST.empty insns in
6 let g = G.create () in
7 Array.iter (G.add_vertex g) verts;
8 let addterm i = function
9   | Ll.Ret _ | Unreachable -> ()
10  | Br lbl -> G.add_edge g verts.(i) (S.ST.find lbl blocks)
11  | Cbr (_, l1, l2) ->
12    G.add_edge g verts.(i) (S.ST.find l1 blocks);
13    G.add_edge g verts.(i) (S.ST.find l2 blocks)
14  | Switch (_, _, _, lbls) ->
15    let vert l = S.ST.find l blocks in
16    List.map snd lbls |> List.map vert |> List.iter (G.add_edge g verts.(i))
17 in
18 let addedges = function
19   | _, Label _ -> ()
20   | i, Insn _ -> G.add_edge g verts.(i) verts.(i + 1)
21   | i, Term term -> addterm i term
22 in
23 List.iter addedges insns;
24 (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, consider 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 definitions, 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:

$$in[n] = use[n] \cup (out[n] - def[n]) \quad (1)$$

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:

$$out[n] = \bigcup_{s \in succ[n]} in[s] \quad (2)$$

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

```

```

3
4 let use (s : S.SS.t) (insn : Cfg.insn) =
5   let op o s = match o with Ll.Id i -> S.SS.add i s | _ -> s in
6   let po = Fun.flip op in
7   match insn with
8   | Insn (_, AllocaN (_, (_, o))) (* | Bitcast _ | ... | Zext _ *) ->
9     op o s
10  | Insn (_, Binop (_, _, l, r)) (* | Icmp _ | Store _ *) ->
11    op l s |> op r
12  | Insn (_, Call (_, _, args)) -> List.map snd args |> List.fold_left po s
13  | Insn (_, Gep (_, bop, ops)) -> List.fold_left po (op bop s) ops
14  | Insn (_, Select (c, (_, l), (_, r))) -> op c s |> op l |> op r
15  | Insn (_, PhiNode (_, ops)) -> List.map fst ops |> List.fold_left po s
16  | Term (Ret (_, Some o) | Cbr (o, _, _)) -> op o s
17  | _ -> 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:

```

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

```

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

```

1 let flowout (i, _) =
2   let newout =
3     let succ = Cfg.G.succ g ids.(i) in
4     List.fold_left
5       (fun s v -> S.SS.union s in_.(Cfg.G.V.label v))
6       S.SS.empty succ
7   in
8   let changed = not (S.SS.equal newout out.(i)) in
9   if changed then out.(i) <- newout;
10  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:

```

1 let dataflow (insns : Cfg.insn list) (ids : Cfg.G.V.t array) (g : Cfg.G.t) =
2   let insns = List.mapi (fun i v -> (i, v)) insns |> List.rev in
3   let in_ = Array.init (List.length insns) (fun _ -> S.SS.empty) in
4   let out = Array.init (List.length insns) (fun _ -> S.SS.empty) in
5   let rec dataflow () =
6     let flowout = (* ... *)
7     let flowin = (* ... *)
8     let flow changed insn = changed || flowout insn || flowin insn in
9     if List.fold_left flow false insns then dataflow () else (in_, out)
10  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.

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     for (int i = 1; i <= argc; i++) {
6         printf("%d!\n", i);
7     }
8
9     return 0;
10 }

```

Figure 10: `cat tests/count.ll`

```

1 define i32 @main (i32 %0, i8* %1) {
2     %3 = icmp slt i32 %0, 1
3     br i1 %3, label %4, label %5
4:
5     ret i32 0
6:
7     %6 = phi i32 [%8, %5], [1, %2]
8     %7 = call i32 @printf (i8*, ...) @printf (i8* @.str, i32
9     %8 = add i32 %6, 1
10    %9 = icmp eq i32 %6, %0
11    br i1 %9, label %4, label %5
12 }

```

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 halved 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	0,6	0,8
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
0	0	3	0	3	0	3	0	3,8	0,8	3,8	0,8	3,8	0,8	3,8	0,8	3,8
1	3		3	8	3,8	8	3,8	8	3,8	8	3,8	8	3,8	8	3,8	8
2																
3																
4				8	8	8	8	8	8	8	8	8	8	8	8	8
5	8	6	8	6	8	6	8	6	8	0,6	0,8	0,6	0,8	0,6	0,6	0,8
6	6	7	6	6	6	6	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	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
9	9		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

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
36,11s user 0,06s system 99% cpu 36,213 total
$ time dune exec build -- tests/sha256.ll -t lva
18,02s user 0,05s system 99% cpu 18,090 total
```

Figure 12: Runtime over single representative run of liveness analysis

## 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

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

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

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

and for any other instruction that defines a variable  $a$

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

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:

```
1 let vert get =
2   match S.ST.find_opt e t with
3   | Some v -> (v, t)
4   | None ->
5     let v = S.SS.add e S.SS.empty |> G.V.create in
6     G.add_vertex g v;
7     (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]. As phi nodes are the only mov instructions, their interference edges are  $\text{out}[n] \setminus \{b_1, \dots, b_n\}$  (see (3)) as opposed to non-move instructions that are just  $\text{out}[n]$ .

```

1 let interf (params : Ll.uid list) (insns : Cfg.insn list) _ (out : S.SS.t array) =
2   let g = G.create () in
3   let vert2 t e = vert g e t |> snd and vert3 e t = vert g e t |> snd in
4   let edge s1 s2 t =
5     let v1, t = vert g s1 t in
6     let v2, t = vert g s2 t in
7     if v1 <> v2 then G.add_edge g v1 v2;
8     t
9   in
10  let t = List.fold_left vert2 S.ST.empty params in
11  let param t p1 =
12    let param t p2 = if p1 <> p2 then edge p1 p2 t else t in
13    List.fold_left param t params
14  in
15  let t = List.fold_left param t params in
16  let setverts t s = S.SS.fold vert3 s t in
17  let t = List.map (def S.SS.empty) insns |> List.fold_left setverts t in
18  let ids = function Ll.Id id -> Some id | _ -> None in
19  let defoutedges t (i, n) =
20    let defs = def S.SS.empty n in
21    let out =
22      match n with
23      | Cfg.Insn (_, Ll.PhiNode (_, ops)) ->
24        List.map fst ops |> List.filter_map ids |> S.SS.of_list
25        |> S.SS.diff out.(i)
26      | _ -> out.(i)
27    in
28    let outedge e1 t = S.SS.fold (edge e1) out t in
29    S.SS.fold outedge defs t
30  in
31  let t = List.mapi (fun i n -> (i, n)) insns |> List.fold_left defoutedges t in
32  (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

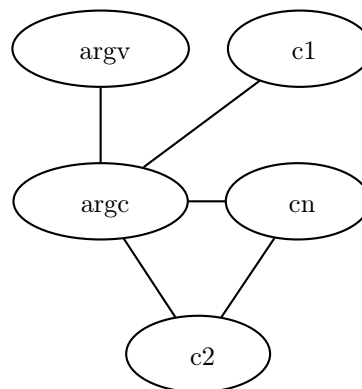


Figure 15: dune exec build -- -t dot

## 5 Graph Coloring

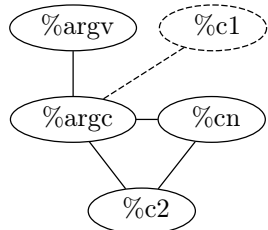
Once an interference graph is constructed, the actual assignments can be found using graph coloring. Graph coloring is a problem in graph theory where the goal is to assign colors to the vertices of a graph in such a way that no adjacent vertices are assigned the same color. A graph that can be colored 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 simplification 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  $\leq 2$ . This wouldn't be a problem on modern day processors most of which have  $\sim 16$  or so GPRs, but for the sake of clearness one can imagine a register machine of  $k = 2$  working registers.

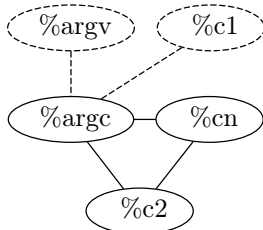
If at first a single node of insignificant degree (i.e.  $< 2$ ) is selected, it is then removed and pushed to a stack. In Figure 15 only `%argv` and `%c1` are of insignificant degree, so either of those is chosen to be removed (represented with dashed borders and edges) as can be seen on Figure 16a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



(a) Graph

`%c1`

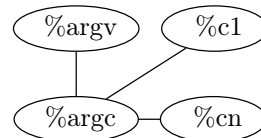
(b) Stack



(a) Graph

`%argv`  
`%c1`

(b) Stack



(a) Graph

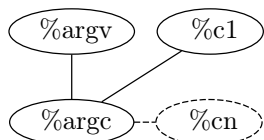
(b) Stack

Figure 16: Push `%c1`

Figure 17: Push `%argv`

Figure 18: Spill `%c2`

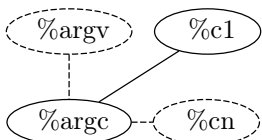
From this point on nothing can be done to further simplify the graph as all of the remaining nodes are of significant degree ( $\geq 2$ ). Therefore, one of the remaining nodes is spilled to memory and entirely removed from the graph as can be seen on Figure 18a and the stack reset as can be seen on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



(a) Graph

`%cn`

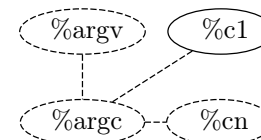
(b) Stack



(a) Graph

`%argv`  
`%cn`

(b) Stack



(a) Graph

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

(b) Stack

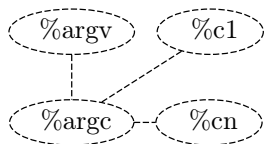
Figure 19: Push `%cn`

Figure 20: Push `%argv`

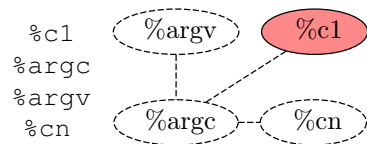
Figure 21: Push `%argc`

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.

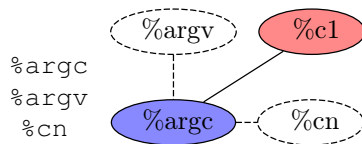


(a) Graph



(b) Stack

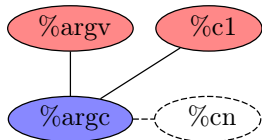
Figure 23: Pop `%c1`



(b) Stack

Figure 24: Pop `%argc`

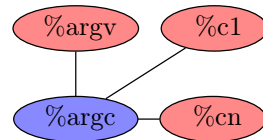
As the last variable `%c1` is pushed in Figure 22 and an empty graph remains, the process of reconstructing the graph can begin. When popping `%c1` in Figure 23, it can be assigned any of the two colors as it has no neighbors adjacent, so either is chosen. On the other hand, when assigning `%argc`, it's 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

$$\begin{aligned} \text{assign}[\%argv] &= \text{assign}[\%c1] = \text{assign}[\%cn] = \%red \\ \text{assign}[\%argc] &= \%blue \end{aligned}$$

and the spilled variable `%c2`

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

## 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  $\leq K$  neighbors 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

( $deg(\%cn) = 2 \geq 2$  and  $deg(\%argc) = 3 \geq 2$ ). 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.

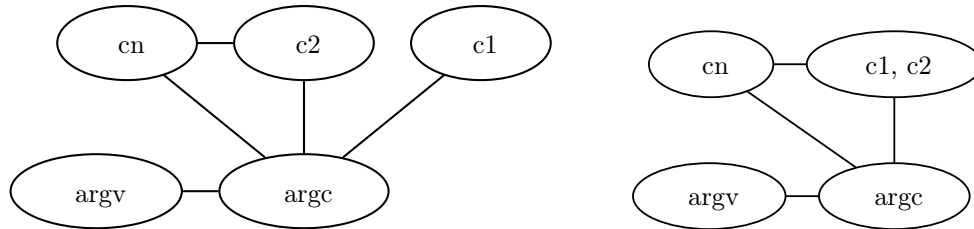


Figure 27: `build -- -t dot -a briggs -n 2`      Figure 28: `-a briggs -n 3`

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

### 5.3 Implementation

The first step is deriving all preference edges:

```

1  let prefer (insns : Cfg.insn list) : S.SS.t S.ST.t =
2    let insn t = function
3      | Cfg.Insn (Some d, Ll.PhiNode (_, ops)) ->
4        List.fold_left
5          (fun t o ->
6            match o with
7              | Ll.Id sop, _ ->
8                S.ST.update sop
9                  (function
10                     | Some s -> Some (S.SS.add d s)
11                     | None -> Some (S.SS.singleton d))
12                    t
13              | _ -> t)
14          t ops
15      | _ -> t
16    in
17    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 instructions don't make sense to eliminate.

Briggs coalescing is then implemented as follows:



```

1  let coalesce_briggs k (prefs : S.SS.t S.ST.t)
2    ((l, g) : Lva.G.V.t S.ST.t * Lva.G.t) : Lva.G.V.t S.table * Lva.G.t =
3    let try_coalesce sop dop (l, g) =
4      if not (Lva.G.mem_edge g (S.ST.find sop l) (S.ST.find dop l)) then
5        let sneighs = S.ST.find sop l |> Lva.G.succ g |> Lva.VS.of_list
6          and dneighs = S.ST.find dop l |> Lva.G.succ g |> Lva.VS.of_list in
7        let neighs = Lva.VS.union sneighs dneighs in
8        let sign v = Lva.G.succ g v |> List.length >= k in
9        let signeighs = Lva.VS.filter sign neighs in
10       if Lva.VS.cardinal signeighs < k then
11         coalesce (S.ST.find sop l) (S.ST.find dop l) (l, g)
12       else (l, g)
13     else (l, g)
14   in
15   let try_pref sop dops (l, g) = S.SS.fold (try_coalesce sop) dops (l, g) in
16   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  $\Phi$ -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.

```

1  let coalesce v1 v2 (st, g) =
2    if v1 = v2 then (st, g)
3    else
4      let g' = Lva.G.create () in
5      let st', vt', _ =
6        Lva.G.fold_vertex
7          (fun v (st, vt, jn) ->
8            if v = v1 || v = v2 then (
9              match jn with
10             | None ->
11               let s = Lva.G.V.label v in
12                 (st, vt, Some (v, s))
13             | Some (v1, s1) ->
14               let s' = Lva.G.V.label v |> S.SS.union s1 in
15                 let v' = Lva.G.V.create s' in
16                   Lva.G.add_vertex g' v';
17                 ( S.SS.fold (fun e l -> S.ST.add e v' l) s' st,
18                   Lva.VT.add v1 v' vt |> Lva.VT.add v v',
19                   None ))

```

```

20     else
21         let s = Lva.G.V.label v in
22         let v' = Lva.G.V.create s in
23         Lva.G.add_vertex g' v';
24         ( S.SS.fold (fun e st -> S.ST.add e v' st) s st,
25           Lva.VT.add v v' vt,
26           jn ))
27     g
28     (S.ST.empty, Lva.VT.empty, None)
29 in
30 Lva.G.iter_edges
31 (fun v1 v2 ->
32   let v1 = Lva.VT.find v1 vt' in
33   let v2 = Lva.VT.find v2 vt' in
34   if v1 <> v2 then Lva.G.add_edge g' v1 v2)
35 g;
36 (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 depends on the context in which it is used.

While compilers like clang generally has an end-goal of machine code, languages like C# and Java will target a VM instead. 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 is the fact that VM bytecode needs to be processed beforehand. While bytecode is generated ahead-of-time, running it on a target machine by interpreting such instructions in real time is inefficient. Therefore, bytecode is generally translated to native machine code immediately prior to being executed, a process known as *just-in-time* (JIT) compilation.

A significant downside of JIT is the fact it needs to be compiled in full before execution can begin, hence processes such as register allocation are deferred to runtime. Therefore, generally a faster class of allocators are used, 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 liveness, 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 with  $k$  variables already active, 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.

Total time spent during compilation is somewhere around those listed in Figure 32.

```

$ time dune exec build -- tests/sha256.ll -a greedy
38,72s user 0,07s system 99% cpu 38,833 total
$ time dune exec build -- tests/sha256.ll -a linear
38,72s user 0,04s system 99% cpu 38,813 total
$ time dune exec build -- tests/sha256.ll -a simple
120,55s user 0,07s system 99% cpu 2:00,74 total
$ time dune exec build -- tests/sha256.ll -a briggs
119,86s user 0,05s system 99% cpu 2:00,03 total

```

Figure 32: Runtime over single runs of compilation

## 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 34 and its associated live ranges/intervals of Figure 33. 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.

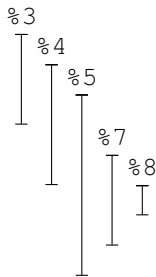


Figure 33: Live intervals

```

1  define i32 @main (i32 %0, i8* %1) {
2      %3 = getelementptr i8*, i8** %1, i64 1
3      %4 = load i8*, i8** %3
4      %5 = call i32 @atoi (i8* %4)
5      call i32 @printf (i8* @.str, i8** %3, i8* %4, ...
6      %7 = call i32 @isqrt (i32 %5)
7      %8 = call i64 @strtoll (i8* %4, i8* null, i32 10)
8      call i32 @printf (i8* @.str.1, i32 %5, i64 %8)
9      call i32 @printf (i8* @.str.2, i32 %7, i32 %5)
10     ret i32 %5
11 }

```

Figure 34: @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, my approach relies on two passes: one for finding the start points (see Figure 35), the other for finding the end points.

```

1 let add e = function
2   | Some s -> Some (S.SS.add e s)
3   | None -> Some (S.SS.singleton e)
4 let intervalstart insns (in_, out) =
5   let insn (ordstarts, starts, active) (i, _n) =
6     let out' = S.SS.diff (S.SS.union in_.(i) out.(i)) active in
7     ( S.SS.fold (fun e a -> IT.update i (add e) a) out' ordstarts,
8       S.SS.fold (fun e a -> S.ST.add e i a) out' starts,
9       S.SS.union active out' )
10  in
11 List.fold_left insn (IT.empty, S.ST.empty, S.SS.empty) insns

```

Figure 35: 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.

```

1 let intervalends insns (in_, out) =
2   let insn (i, _n) (ordends, ends, active) =
3     let in' = S.SS.diff (S.SS.union in_.(i) out.(i)) active in
4     ( S.SS.fold (fun e a -> IT.update i (add e) a) in' ordends,
5       S.SS.fold (fun e a -> S.ST.add e i a) in' ends,
6       S.SS.union active in' )
7   in
8   List.fold_right insn insns (IT.empty, S.ST.empty, S.SS.empty)

```

Figure 36: Implementation of intervalendsintervalstart in lib/linear.ml

The opposite, i.e. finding the interval ends, is quite similar (see Figure 36), with the only real difference being the use of `List.fold_right` instead of `List.fold_left` to reduce the function body in reverse.

```

1 let rec linearscan insns spills (avail, active, assign, incstart, incend) =
2   match IT.min_binding_opt incstart with
3   | Some (i, ss) ->
4     let e = S.SS.choose ss in
5     let ss' = S.SS.remove e ss in
6     let incstart' =
7       if S.SS.is_empty ss' then IT.remove i incstart
8       else IT.add i ss' incstart
9     in
10    let spills, (avail', active', assign', incstart', incend') =
11      match Regs.choose_opt avail with
12      | Some reg ->
13        ( spills,
14          expire i
15            ( Regs.remove reg avail,
16              S.ST.add e reg active,
17              S.ST.add e (Reg reg) assign,

```

```

18         incstart',
19         incend ) )
20 | None ->
21     let _j, spill =
22         IT.filter
23             (fun _k v -> S.SS.exists (fun e -> S.ST.mem e active) v)
24             incend
25         |> IT.max_binding
26     in
27     let spill =
28         S.SS.filter (fun e -> S.ST.mem e active) spill |> S.SS.choose
29     in
30     let reg = S.ST.find spill active in
31     let incend' =
32         IT.filter_map
33             (fun _ ss ->
34                 let ss' = S.SS.remove spill ss in
35                 if S.SS.is_empty ss' then None else Some ss')
36             incend
37     in
38     ( spills + 1,
39       expire i
40         ( avail,
41           S.ST.remove spill active |> S.ST.add e reg,
42           S.ST.remove spill assign
43             |> S.ST.add spill (stack spills)
44             |> S.ST.add e (Reg reg),
45           incstart',
46           incend' ) )
47 in
48     linearscan insns spills (avail', active', assign', incstart', incend')
49 | None -> expire (List.length insns) (avail, active, assign, incstart, incend)

```

Figure 37: Implementation of linearscan in linear.ml

The implementation of the actual linearscan iteration in Figure 37 is quite tedious, but the core idea is the outermost match block. The smallest interval is selected using `min_binding_opt`. This is done to visit each interval in increasing order of start point. If there are none left, the linearscan iteration is essentially over, but calls `expire` regardless to clean things up. The case of intervals remaining is more interesting, as one of the variables starting at  $i$  needs to be assigned. First a variable is chosen, the order of this is not important. Despite the minimal example used in Figure 34, there are cases where several intervals begin at the same point (e.g. loops), hence a symbol set is used. Lines 6-8 are just concerned with removing the chosen variable from the table of increasing start points. Then, an available register is chosen. If one is available (line 12), it is 1) removed from the set of available registers, 2) added to the symbol table of active variables, and 3) assigned the same register in the returned assignments table. If no registers are available on the other hand (line 20), a variable needs to be spilled. It does this by first filtering through and only visiting the end points that are currently active (i.e. members of `active`). Then the one of intervals that are currently active with the greatest end points is chosen by first filtering end points for which at least of the variables is active (lines 22-25), and then selecting one of those immediately after (line 28). The currently assigned register of that variable is then looked up (line 30) and the end point is removed (lines 32-36).

In the case of spilling, the idea is to 1) remove the previously assigned variable with the greatest remaining length from the active set, 2) assign the previously assigned variable to the nextmost available stack slot, and 3) assign the currently visited start interval to the register just unassigned from the spilled variable.

In either of those cases, the `expire` function is called right after. This is to ensure that there are no dangling variables left in the table of increasing end points, and that their previously assigned registers are put back in the available set of registers as can be seen in Figure 38.

```

1 let rec expire i (avail, active, assign, incstart, incend) =
2   match IT.min_binding_opt incend with
3   | Some (j, end_) ->
4     if j >= i then (avail, active, assign, incstart, incend)
5     else
6       let s = S.SS.choose end_ in
7       let reg = S.ST.find s active in
8       let active' = S.ST.remove s active in
9       let avail' = Regs.add reg avail in
10      let incend' = IT.remove j incend in
11      expire i (avail', active', assign, incstart, incend')
12 | None -> (avail, active, assign, incstart, incend)

```

Figure 38: Implementation of `expire` in `lib/linear.ml`

Putting it all together, is the `alloc` function in `lib/linear.ml` that builds the set of available registers from the  $k$  passed, builds the increasing interval starts and ends tables, and finally, begins the iterative process of linear scan.

```

1 let alloc k insns (in_, out) =
2   let avail = List.init k reg_of_int |> Regs.of_list in
3   let insns = List.mapi (fun i n -> (i, n)) insns in
4   let incstart, _, _ = intervalstart insns (in_, out) in
5   let incend, _, _ = intervalends insns (in_, out) in
6   let _, _, asn, _, _ =
7     linearscan insns 0 (avail, S.ST.empty, S.ST.empty, incstart, incend)
8   in asn

```

Figure 39: Implementation of `alloc` in `lib/linear.ml`

## 7 Instruction Selection

### 7.1 LLVM-- instruction set

The instruction set used in this paper will be a combination of those 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 addition 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 concrete 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 type noted 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
<code>add</code>	<code>addx %src, %dst</code>	{}
<code>alloca</code>	<code>subq \$n, %rsp</code>	{%rsp}
<code>alloca_n</code>	<code>subq \$n, %rsp</code>	{%rsp}
<code>and</code>	<code>andx %src, %dst</code>	{}
<code>ashr</code>	<code>sarx %src, %dst</code>	{}
<code>bitcast</code>	<code>movx %src, %dst</code>	{}
<code>call</code>	<i>see 7.2.1</i>	{%rax, %rcx, %rdx, %rsi, %r8-%r11}
<code>gep</code>	<i>see 7.2.2</i>	{}
<code>icmp</code>	<code>cmpx, set</code>	{%rFLAGS}
<code>load</code>	<code>movx (%src), %dst</code>	{}
<code>lshr</code>	<code>shrx %src, %dst</code>	{}
<code>mul</code>	<code>imulx %src, %dst</code>	{%rax}
<code>or</code>	<code>orx %src, %dst</code>	{}
<code>phi</code>	<i>see 7.2.3</i>	{}
<code>ptrtoint</code>	<code>movx %src, %dst</code>	{}
<code>sdiv</code>	<code>idivx %src, %dst</code>	{%rax, %rdx}
<code>select</code>	<code>cmpx, cmovex</code>	{%rFLAGS}
<code>sext</code>	<code>movx %src, %dst</code>	{}
<code>shl</code>	<code>shl %src, %dst</code>	{}
<code>srem</code>	<code>idivx %src, %dst</code>	{%rax, %rdx}
<code>store</code>	<code>movx %src, (%dst)</code>	{}
<code>sub</code>	<code>subx %src, %dst</code>	{}
<code>trunc</code>	<code>movx %src, %dst</code>	{}
<code>udiv</code>	<code>divx %src, %dst</code>	{%rax, %rdx}
<code>urem</code>	<code>divx %src, %dst</code>	{%rax, %rdx}
<code>xor</code>	<code>xorx %src, %dst</code>	{}
<code>zext</code>	<code>movx %src, %dst</code>	{}

Table 3: LLVM-- instruction to x86 translation table

### 7.2.1 Translating `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 functions, only those where the registers 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 at least one, 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 Translating `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.

<pre> 1  typedef struct { 2      char c; 3      int x; 4      int y; 5  } vector; 6 7  void dot(vector *a, vector *b) { 8      a-&gt;x += b-&gt;x; 9      a-&gt;y += b-&gt;y;    // -----&gt; 10 } </pre>	<pre> 1  ; ... 2  %6 = getelementptr @struct.vector, 3      %i8* %1, i64 0, i32 2 4  %7 = load i32, i32* %6 5  %8 = getelementptr @struct.vector, 6      %i8* %0, i64 0, i32 2 7  %9 = load i32, i32* %8 8  %10 = add i32 %9, %7 9  store i32 %10, i32* %8 10 ; ... </pre>
---	--

Figure 40: Subfield access

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

Essentially `gep` instructions allow for unbounded array lookups, in that a `getelementptr` instruction is followed by the base address and one or more scaled offsets. What happens in Figure 41 is `vector` type is reduced to an `int` array, because it essentially is: in C, struct fields are generally padded such that their respective memory addresses are aligned on four- or eight-byte boundaries [17]. As such, accessing the third field entry corresponds to accessing the third element of an `int` array. The `getelementptr` instruction gets the pointer of this element by multiplying the size with the index and adding that to the base address. Therefore, the x86 that is generated essentially amounts to an `imul` instruction as can be seen in Figure 42.

```

1  ; ...
2  movq %rdi, %rcx
3  movq $0, %rax
4  addq %rax, %rcx
5  imulq $8, %rax
6  movq $2, %rax
7  imulq $4, %rax
8  addq %rax, %rcx
9  ; ...

```

Figure 42: `dune exec build -- -t x86`

where the base register `%1` has assignment `assign[%1] = %rdi` is moved into the scratch register `%rcx`, and each of the offset operands are 1) moved into the other scratch register `%rax`, 2) multiplied by the size of their respective types, and 3) added to the base register.

Unfortunately, `gep` wasn't implemented properly in time for `handin`. Instead the code appears to be left over from what was jumbled together in the Tiger days, so it does work in *some* cases but far from all.



### 7.2.3 Translating `phi` instructions

Another irregular instruction is the `phi` instruction representing the  $\Phi$ -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[bi]` 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 43: `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 44: `dune exec build -- -t x86`

Notice for instance Figure 43 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 44. 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 preparation for continuing execution at `_main$3`.

As `mov` operations do not change the `FLAGS` register [18], having `mov` instructions between the branching instructions of `je` and `jmp` do not affect the branching. Likewise, because of the `use[n]` semantics of  $\Phi$ -functions, all `bi` are in interference in one another, so may not be overwritten by the `movs` inserted.

One particularly nasty oversight that was discovered a couple of days prior to `handin` stems from phi `movs` overwriting each other. Consider Figure 45 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 `%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, %11 = phi i32 [%0, %1], %0], [%0, %1]
10 ret i32 %11
11 }

```

Figure 45: cat tests/sqrt1.ll

```

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 46: buiild -- -t x86

This was resolved by relying on conditional movs immediately prior to the conditional branch (i.e. `je`). Specifically, the `cmovq` instruction was used, which only moves from source to destination if the equality flag (i.e. ZF) is set. Unfortunately the `cmovq` 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 48.

```

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

```

Figure 47: buiild -- -t x86

```

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  cmovq   %rax, %rcx
8  movq    %rcx, -16(%rbp)
9  je      _main$9
10 movq    -16(%rbp), %rax
11 # ...

```

Figure 48: buiild -- -n 0 -t x86

### 7.2.4 Translating terminators

As for implementing terminators, the simplest is `unreachable`, as this literally doesn't evaluate to anything. Next is `ret`, which is more involved, in that it needs to concern itself with calling conventions. Although these are fairly straight forward, especially since functions generally apply the same operations in the prologue by pushing all callee-saved registers to the stack and then, occasionally, subtracting the amount of spilled variables from `%rsp`. The function epilogue doesn't have to concern itself with the number of spilled variables though, as the stack pointer is saved to `%rbp` and simply restored from that. The return value (if specified) is moved to the return register which is `%rax` according to the System V AMD64 ABI [13, p. 26].

When it comes to branching, whether conditional or not, the phi nodes need to be considered. As explained in 7.2.3, the purpose of phi instructions is to ensure values are moved into assigned registers immediately *before* branching, hence phi nodes are looked up and inserted before `jmp` instructions. Unfortunately this wasn't taken into account for `switch` instructions, but should be fine for both `br` instructions.

## 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/framework

The test suite is quite extensive although not anywhere close to complete. Some tests have been handwritten during development, mostly meant to assert one specific feature (unit tests), but others are also intended 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 `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 successfully 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 20% of the LLVM test suite that were never working.

### 7.4.1 Debug Harness

A more elaborate way of asserting correctness of translation is to assert that there are no effects except for those intended. One way of achieving this is to check if a sequence of x86 instructions representing a single LLVM-- operation change any values except those expected. An approach to this is to insert a breakpoint before every LLVM-- instruction, then attach a debugger that breaks on these, and ensure that only the intended registers are altered.

To achieve this, only a few changes need to be made to the codebase. A debug flag is introduced, which inserts a label before every instruction by adding a `Breakpoint` variant of the `ins` type before the assembly emitted of that specific operation. A unique identifier as well as a list of changes expected to be made is encoded. This includes a bitmask of GPRs that can possibly be changed (aside from the scratch registers of course) as well as which parts of the stack may be modified. In addition to this, each of the labels are output with a `.globl` directive in the file header. A file is built with these extra symbols using the `-d` flag.

Unfortunately, this was never made to work in practice, but the `harness.py` script is attached regardless. It was made and tested with `clang16` in mind on Linux.

## 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 parameter 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.

All benchmarks are performed on a Ryzen 9 5900x.

### 8.1.1 `benches/fib.ll`

The 'dumb' fib is a very naive approach to finding the  $n$ th 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 1024`

The measures in Table 4 are remarkably similar 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` instead.

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 5:

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 1024`

While it's not a complete one to one 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/loopn.ll`

Another simple example is a program that only consists of a loop running for  $n$  iterations. These stem from the `tests/loopn.c` file but with the call to `printf` removed such that no unnecessary time is spent doing I/O. Depending on the level of optimization chosen, either the mutation strategy of stack allocated variables or phi nodes are used. For the case of using the stack, the simplicity of such a program shines through as can be seen on Table 6:

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
1048575	2238539	2.56041x	2.63938x	2.56101x	2.63915x	2.56111x	19.26926x	27.70065x	36.60162x
2097151	4333585	2.61339x	2.69416x	2.61334x	2.69427x	2.61352x	19.87471x	28.58590x	37.78097x
4194303	8529189	2.63937x	2.72120x	2.63934x	2.72124x	2.63938x	20.17961x	29.03173x	38.37560x
8388607	16919352	2.65269x	2.73526x	2.65275x	2.73534x	2.65264x	20.33697x	29.26211x	38.68258x
16777215	33697415	2.65966x	2.74260x	2.65966x	2.74263x	2.65964x	20.41817x	29.39601x	38.84065x
33554431	67254339	2.66311x	2.74627x	2.66312x	2.74628x	2.66311x	20.45872x	29.46374x	38.91968x
67108863	134365449	2.66491x	2.74816x	2.66488x	2.74816x	2.66492x	20.47947x	29.49845x	38.96011x

Table 6: Benchmark of `benches/factori64.ll` output by `dune exec bench -- -f factori32 -n 1024`

Despite having no optimizations enabled and using the stack to store the counter, clang manages to build an executable wherein the loop consists of no more than 7 instructions (see Figure 49). In comparison, mine

is 19 (see Figure 50):

```

# ...
movslq 0xc(%rsp),%rax
cmp 0x10(%rsp),%rax
jae 0x1199 <main+89>
mov 0xc(%rsp),%eax
add $0x1,%eax
mov %eax,0xc(%rsp)
jmp 0x1179 <main+57>
# ...

# ...
movq %rsi, %rax
movl (%rax), %eax
movq $0, %rdx
movl %eax, %edx
movq %rdx, %rax
movq $0, %rbx
movl %eax, %ebx
movq %rdi, %rax
movq (%rax), %rax
movq %rax, %rdx

# continued...
movq %rbx, %rax
movq %rdx, %rcx
cmpq %rcx, %rax
movq $0, %rdx
setb %dl
movq %rdx, %rax
cmpq $0, %rax
je _main$20
jmp _main$17
# ...

```

Figure 49: clang

Figure 50: build -- -t x86

It does make sense that my compiler nearly takes 3x as many cycles considering that nearly 3x as many instructions are being executed. Because of the way values are stored, there is literally no benefit from Briggs coalescing since no phi nodes are involved ever. I was surprised to see Greedy and Linear 2 to be that much slower, but on second thought it makes perfect sense since loading variables from the stack before every single arithmetic/logic operation is considerably slower. What's more surprising is the fact that Tiger is nearly 4x faster than any Greedy (see Table 7). It's presumably just because all binary operations consist of about three instructions or so.

arg(s)	clang	greedy 12	greedy 8	greedy 6	greedy 4	greedy 2	greedy 1	greedy 0	tiger
1048575	2237215	27.71735x	36.15447x	36.62296x	36.62342x	36.62283x	36.62360x	36.62324x	10.97018x
2097151	4334414	28.58063x	37.28996x	37.77377x	37.77391x	37.77373x	37.77353x	37.77356x	11.29302x
4194303	8529224	29.03182x	37.88371x	38.37535x	38.37541x	38.37546x	38.37544x	38.37543x	11.46122x
8388607	16919774	29.26144x	38.18576x	38.68170x	38.68169x	38.68175x	38.68169x	38.68163x	11.54653x
16777215	33698040	29.39364x	38.34210x	38.83996x	38.83997x	38.83991x	38.83995x	38.83993x	11.59127x
33554431	67253512	29.46538x	38.42125x	38.92015x	38.92019x	38.92020x	38.92016x	38.92020x	11.61419x
67108863	134364289	29.49788x	38.46111x	38.96041x	38.96050x	38.96052x	38.96043x	38.96040x	11.62584x

Table 7: Benchmark of benches/loopn0.ll output by dune `exec bench -- -f loopn0 -n 64`

Even with `-O1` where phi nodes are introduced, does the Briggs allocator seem to affect anything as can be seen in Table 8. This is because the only two variables eligible for coalescing, `%9` and `%10`, are in interference as both are used at intersecting program points.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
131071	401485	1.65252x	1.65413x	1.65379x	1.65354x	1.65376x	1.65265x	1.63694x	8.19401x
262143	663679	1.79023x	1.79079x	1.78950x	1.79092x	1.78914x	1.78887x	1.79096x	9.69501x
524287	1188029	1.88330x	1.88252x	1.88192x	1.88240x	1.88242x	1.88211x	1.88284x	10.79175x
1048575	2237256	1.93727x	1.93740x	1.93674x	1.93715x	1.93728x	1.93739x	1.93690x	11.42118x
2097151	4334546	1.96774x	1.96724x	1.96767x	1.96746x	1.96749x	1.96756x	1.96753x	11.81912x
4194303	8529268	1.98350x	1.98348x	1.98353x	1.98339x	1.98339x	1.98358x	1.98355x	12.04037x
8388607	16918996	1.99159x	1.99166x	1.99166x	1.99160x	1.99166x	1.99165x	1.99162x	12.13501x

Table 8: Benchmark of benches/loopn1.ll output by dune `exec bench -- -f loopn1 -n 64`

### 8.1.3 benches/phis.11

To actually see the benefit of coalescing in action I have introduced some generated tests of benches/phis{0-7}.11. Generated with benches/phis.py <n> <m> that produces a .11 file that consists of a loop hardcoded for  $n$  repetitions with  $m$  phi nodes.

phis	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
0	1073928603	2.49981x	2.49981x	1.24997x	1.24997x	2.49981x	2.49982x	2.49981x	11.68360x
1	1073928952	2.54982x	2.60344x	1.49994x	1.49994x	2.49981x	2.49983x	2.54981x	11.58368x
2	1610830251	1.75897x	1.99991x	1.16665x	1.33330x	1.66660x	1.66663x	1.75644x	8.40916x
3	2147723548	1.74997x	1.87495x	1.05002x	1.06250x	1.24999x	1.49997x	1.53576x	6.64460x
4	2684621503	1.40001x	1.79998x	1.00000x	1.00000x	1.00000x	1.29999x	5.43928x	5.99128x
5	3221532192	1.49997x	1.83329x	1.00000x	1.00000x	0.99999x	1.24999x	4.60720x	5.82084x
6	3758430778	1.35714x	1.78839x	0.88572x	0.90478x	0.85715x	1.21428x	4.10980x	5.54983x
7	4295303796	1.50000x	1.81439x	0.87500x	0.87502x	0.87500x	1.18750x	3.62761x	4.84205x

Table 9: Benchmark of benches/phis{0..7}.11 output by dune exec bench -- -f phis -n 16

### 8.1.4 benches/factori32.11

Integer factorization is another interesting algorithm. Generally considered to be NP, here are the results of factorizing known primes of increasing bit size seen in Table 10

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
16777213	168400	1.04991x	1.02776x	1.04829x	1.04435x	1.04465x	1.82795x	2.31432x	2.32746x
33554393	175723	1.06090x	1.03777x	1.06480x	1.04465x	1.03702x	2.12225x	2.78015x	2.79323x
67108859	185913	1.05642x	1.08067x	1.08352x	1.07946x	1.08349x	2.50046x	3.38055x	3.39737x
134217689	199433	1.11084x	1.11393x	1.08952x	1.11192x	1.10441x	2.97084x	4.13960x	4.12783x
268435399	217520	1.14321x	1.15268x	1.13781x	1.15059x	1.15174x	3.56567x	5.06899x	5.10944x
536870909	242651	1.19742x	1.19637x	1.18878x	1.19987x	1.19055x	4.25606x	6.18250x	6.21937x
1073741789	282883	1.23359x	1.23140x	1.23124x	1.21433x	1.23208x	4.93689x	7.23489x	7.31718x

Table 10: Benchmark of benches/factori32.11 output by dune exec bench -- -f factori32 -n 1024

What's more interesting is that increasing the input size actually reduces the discrepancy with the Greedy and Linear 2 allocators, while all others seem to rise at the same pace as can be seen in Table 11.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
268435399	218872	1.06367x	1.10706x	1.06818x	1.10527x	1.10276x	1.63497x	2.38428x	2.38607x
536870909	246634	1.08777x	1.12660x	1.08040x	1.13337x	1.13368x	1.79734x	2.73281x	2.74118x
1073741789	285516	1.09813x	1.16160x	1.10120x	1.16364x	1.16724x	1.97253x	3.11594x	3.12868x
2147483647	340871	1.12733x	1.19431x	1.12500x	1.19348x	1.18683x	2.15246x	3.49642x	3.51766x
4294967291	419568	1.14194x	1.21527x	1.14432x	1.21632x	1.22207x	2.32057x	3.87806x	3.89161x
8589934583	529755	1.16337x	1.24534x	1.16161x	1.24781x	1.25015x	2.47971x	4.22521x	4.22809x
17179869143	682438	1.18197x	1.28018x	1.18192x	1.27765x	1.27933x	2.63014x	4.54643x	4.56048x
34359738337	907751	1.18869x	1.28923x	1.18944x	1.29097x	1.28925x	2.72756x	4.76366x	4.77551x

Table 11: Benchmark of benches/factori64.11 output by dune exec bench -- -f factori64 -n 1024

### 8.1.5 benches/sieven.11

Sticking to the subject of primes, the Sieve of Eratosthenes is another popular benchmark for gauging runtime performance [19, pp. 243–244]. It is a simple algorithm for generating primes that rely on a runtime allocated array of booleans where the  $i$ th entry represents the primality of  $i$ . This benchmark ought to cover

nested loops and other branching, and as expected the graph coloring based allocators seem to outperform all others. Remarkably Greedy 12 is almost the same as Greedy 0, but this is probably because most operations are performed on an array, for which register assignment has no influence.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
128	157871	1.00777x	1.01034x	1.00764x	1.01089x	1.05785x	1.16044x	1.24422x	1.24352x
256	162153	0.98003x	1.02047x	1.01174x	1.02203x	1.11154x	1.31965x	1.49736x	1.50711x
512	168660	1.02724x	1.04284x	1.03418x	1.04433x	1.22076x	1.61297x	1.97437x	2.03639x
1024	186641	1.02816x	1.05324x	1.04300x	1.05647x	1.38383x	2.15658x	2.77549x	2.89126x
2048	219043	1.07644x	1.10227x	1.07577x	1.10024x	1.65644x	3.04685x	4.07178x	4.24534x
4096	286355	1.11926x	1.15395x	1.11675x	1.15387x	2.00116x	4.19594x	5.76251x	6.03484x
8192	421739	1.15807x	1.20870x	1.15821x	1.20849x	2.36226x	5.43577x	7.57708x	7.91676x
16384	692242	1.19241x	1.25460x	1.19587x	1.25348x	2.66022x	6.50741x	9.15450x	9.54856x

Table 12: Benchmark of benches/sieven.ll output by dune `exec bench -- -f sieven -n 1024`

### 8.1.6 benches/sha256.ll

The sha256 benchmark is a massive program when compared to all others. It stems from an MIT licensed implementation [20]. The core function is `sha256_compress` with no loops, but it does perform 64 SHA-2 rounds which are quite involved by design. As most binary operations depend on five or so instructions instead of 1-2 as is possible for the majority, that alone ought to reduce performance immensely.

arg(s)	clang	simple 12	simple 2	briggs 12	briggs 2	linear 12	linear 2	greedy 12	greedy 0
4	151231	1.24623x	1.28026x	1.24897x	1.28066x	1.85133x	1.88049x	3.00814x	3.02968x
8	159289	1.41012x	1.45279x	1.41295x	1.44789x	2.42693x	2.46595x	4.38812x	4.40305x
16	173067	1.70848x	1.76986x	1.70898x	1.77111x	3.45499x	3.49756x	6.88402x	6.88614x
32	202474	2.15587x	2.25675x	2.14418x	2.26214x	5.03392x	5.12076x	10.69747x	10.70125x
64	256682	2.78746x	2.95124x	2.79279x	2.95499x	7.25491x	7.39688x	16.05447x	16.05908x
128	369901	3.45784x	3.67742x	3.45438x	3.67655x	9.56947x	9.78032x	21.68320x	21.71900x
256	598859	4.01075x	4.27823x	4.01290x	4.27501x	11.53833x	11.80069x	26.47159x	26.47409x
512	1054540	4.40890x	4.70502x	4.40897x	4.70366x	12.93893x	13.22190x	29.86374x	29.85531x
1024	1989472	4.59073x	4.90795x	4.59569x	4.90336x	13.62511x	13.93164x	31.53197x	31.56367x
2048	3831949	4.73191x	5.05794x	4.73420x	5.05587x	14.10539x	14.41266x	32.69551x	32.72633x
4096	7524589	4.80394x	5.13065x	4.80706x	5.13594x	14.33546x	14.66333x	33.25560x	33.30167x
8192	14901775	4.84418x	5.18112x	4.84423x	5.18259x	14.46452x	14.78932x	33.58380x	33.61478x

Table 13: Benchmark of benches/sha256.ll output by dune `exec bench -- -f sha256 -n 256`

## 8.2 Future work

It goes without saying that there are countless optimizations waiting to be made, but as the goal of this project was to explore the different approaches, not many regards were had for the instruction selection itself. In fact, much more concern was taken for making sure each instruction was translated such that no other live variables were affected. The call instruction alone is probably a whole order of magnitude slower than necessary. Likewise, the function pre-/epilogue are also overly cautious.

Certainly, the selection of benchmarks could also have been broader, with the vast majority just showing how Greedy is considerably slower. Not to mention more benchmarks with support for the Tiger compiler that was written the year prior. Likewise, some more detailed profiling of the entire compilation would have been interesting. Some short descriptions of time taking during certain phases are included (see Figure 12 and 32) although a far more detailed report could have been made with instrumentation frameworks like Landmarks [21] as the time spent during compilation is quite significant, particularly for runtime environments that depend on JIT, but also for AOT compilers as long compile times can negatively affect productivity.



As for the liveness analysis, research has been conducted allowing for liveness analysis to be performed over IR of strict SSA form in only two passes [22] shadowing the current approach that depends on iterative dataflow analysis. Similarly, linear scan can be aware of 'holes' in live ranges such that variables don't need to live across widely disjointed basic blocks [23].

## 9 Conclusion

The intended goal of this project has been to implement both the graph coloring and linear scan algorithms and evaluate them both in terms of benefits and drawbacks when used to compile LLVM-- IR to x86 assembly and, where applicable, compare it to the Tiger compiler developed as the primary project during the 2022 compilers course. At first, control flow analysis was implemented to generate a control-flow graph. As this was nearing completion, the different types of allocators were considered. At first the Greedy allocator was implemented, as this is the simplest one and happens to be the one closest to the Tiger compiler. As every variable is assigned its own register/stack slot, no liveness analysis needs to be performed prior to assignment. Then work towards an overly cautious instruction selection began, that would translate a sound sequence of x86 instructions regardless of assignment.

In the meantime, a testing framework was being developed that was intended to compile and execute existing LLVM-- programs and compare their output to the expected output of the `clang` compiler, the output of which has been presumed to be correct. As soon as this was in a somewhat working state, work began on the core of this project: the different sorts of allocators. Both graph coloring and linear scan rely on dataflow analysis which was implemented to derive live ranges for each variable. Some observations were made about the expected runtime of dataflow analysis, after which work began on the graph coloring by simplification as explained in both the Appel and Aho et. al. texts.

Then benchmarks and performance of the different approaches were starting to enter the picture, with the graph coloring already showing to be a considerable improvement when compared with the Greedy allocators, especially when they had no available registers to assign and, much like the Tiger compiler, had to depend on the stack entirely. As the the work on graph coloring was nearing completion, one of the coalescing strategies of the Appel text was also implemented. The resulting Briggs allocator was found to have surprisingly marginal influence on runtime, with only benchmarks written to demonstrate the exact benefits showing any improvement.

With the end of the year approaching fast, the linear scan approach was also implemented. Linear scan is most often used in contexts where the compile-time is critical, so the desired time spent compiling needs to be considerably less when compared to graph coloring. This fortunately turned out to be the case, as the time spent compiling comparatively large programs such as `sha256` were about 1/3 of graph coloring in terms of compile time and was able to keep up with the Greedy allocator despite showing performance gains for the majority of benchmarks.

In the end, the caution taken during instruction selection has definitely been the source of the majority of performance regression when compared to the Tiger compiler, with some benchmarks being upwards of 4x slower, despite fundamentally working with the same assignments. The Simple and Briggs allocators were generally only between 1-2x slower than that of `clang`, with some approaching 5x. The linear scan was surprisingly able to keep up fairly well, but in the case of deep repetition with variables defined early spanning the entire breadth of the loop, and accessed several times within, were considerably slower.

In conclusion, one would not stand to benefit much from reimplementing a well established compiler backend such as LLVM, as the sheer time spent in development means that a far broader set of corner cases are accounted for. Though some execution environments seek different approaches other than the rigorous amount of optimizations made at compile-time to benefit runtime performance and vice versa.

## 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. Ian, 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] LLVM 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 Language 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] Apple Inc. “The powerful programming language that’s also easy to learn.” (), [Online]. Available: <https://web.archive.org/web/20231230182842/https://developer.apple.com/swift/> (visited on 12/30/2023).
- [9] Rust Foundation. “Rust compiler development guide.” (), [Online]. Available: <https://web.archive.org/web/20231230185321/https://rustc-dev-guide.rust-lang.org/overview.html> (visited on 12/30/2023).
- [10] LLVM 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] M. Woodward. “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] Kotlin 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 SIG-PLAN symposium on compiler construction*, ACM, 1982, pp. 98–105, ISBN: 9780897910743.
- [17] “The gnu c reference manual: Structures.” (), [Online]. Available: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Structures>.
- [18] “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>.
- [19] “One million primes through the sieve,” [Online]. Available: <https://archive.org/details/byte-magazine-1985-11/page/n245/mode/2up?view=theater> (visited on 04/19/2016).

- [20] “Fast sha-2 hashes in x86 assembly.” (), [Online]. Available: <https://www.nayuki.io/page/fast-sha2-hashes-in-x86-assembly>.
- [21] “Landmarks: A simple profiling library.” (), [Online]. Available: <https://github.com/LexiFi/landmarks>.
- [22] F. Brandner, B. Boissinot, A. Darté, B. Dupont de Dinechin, and F. Rastello, *Computing Liveness Sets for SSA-Form Programs*, eng. 2011.
- [23] H. Mössenböck and M. Pfeiffer, “Linear scan register allocation in the context of ssa form and register constraints,” eng, in *CC*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 229–246, ISBN: 3540433694.