PARIS
LODRON
UNIVERSITÄT
SALZBURG

Bachelor's Thesis

RISC-U Binary Optimization for Selfie

David Pape

Department of Computer Sciences
Paris Lodron University of Salzburg

Supervisor: Univ.-Prof. Dr.-Ing. Christoph M. Kirsch

December 2021

RISC-U Binary Optimization for Selfie

Abstract

Optimizers are part of any modern compiler, and their practical importance in writing performant code cannot be understated. Still, they introduce major complexity, and its complement: bugs. In this work, we present a binary optimizer for RISC-U, a small subset of RISC-V and the target architecture of selfies[3] educational compiler. To address the complexity issue, our optimizer is structured as a standalone binary, keeping the compiler simple. Since this comes at the cost of compile-time information, a binary analyzer is a prerequisite. With it, our optimizer is able to remove redundant instructions and apply several peephole optimizations, leading to a roughly five percent speedup.

Contents

1 Introduction

Today, an optimizer is part of any production compiler, and massive effort is put into creating better optimizers. When it comes to build times, the optimization stage usually is the largest component, and it also is a major source of compiler bugs. The problems underlying optimization and code analysis are difficult, or in some cases even unsolvable.

The justification for all this effort lies in the fact that optimized code would not be feasible in practice without optimizing compilers. Huge speed gains are commonly hiding away in details specific to the target instruction set, or only expressible in a high level programming language through unintuitive constructs.

For example, the unwieldy expression `a + (a << 3)` implements the multiplication `a * 9` much quicker since it avoids a slow `mul` instruction. What's more, the ISA may offer a few tricks to speed this up even more. In ARM, the whole expression can be implemented with a single instruction: when adding, we may specify a shift offset for one operand that will be applied immediately before the addition.

A stupid enough compiler might not take advantage of this and emit two separate instructions instead. And so, even if programmers forced themselves to write all of their multiplications in this illegible way, the resulting binary would still have obvious potential for optimization.

To realize this potential in a world without optimizing compilers means knowing all about the nooks and crannies of the hardware, and breaking with some of the most fundamental principles of software development. Performance is strongly at odds with the other dimensions of code quality.

Therefore, for any IT department with typical temporal and financial constraints, performant code would be impossible to write if optimizing compilers did not take care of all these minute, yet critical details.

However, the massive complexity and engineering effort inherent to code optimization persists, even if the neat encapsulation into optimizing compilers has, most of the time, made it invisible to developers. The qualifier "most of the time" is necessary because compiler writers are not infallible either. The number of bugs found in GCC has been in the six figures since earlier this year, and while compiler bugs aren't a regularity in practice, they are tough to debug if they do occur.

Hoping to reduce compiler complexity and thereby avoid some of the aforementioned issues, we provide a standalone binary optimizer for RISC-V binaries. The optimizer could be dropped into any toolchain, but is tuned for selfie, an educational compiler, emulator and hypervisor[3]. Due to selfie's educational nature, keeping the compiler simple is an even more important concern to us. By decoupling the optimizer, selfie remains completely untouched.

The optimizer is built on top of a binary analyzer created by Thomas Wulz for his bachelor's thesis project[9]. In the first pass, the analyzer collects information about the binary, which is then used by the optimizer in several optimization passes.

During analysis, we mark instructions that cannot be reached or have no effect on the machine state in every possible execution, as well as collect information about the contents and liveness of registers. In the initial optimization pass, marked instructions are removed. Next, a catalogue of peephole optimizations is employed, replacing inefficient sequences of instructions with faster ones wherever our knowledge about the registers permits it.

When optimizing selfie, we obtain 3125 optimization sites in total, out of 34838 instructions. In a self-compilation of selfie, we obtain a ~5% speed up compared to the unoptimized version. We also profiled the number of effectless instructions at runtime. The optimized version reduces the number of such instructions by roughly 20%.

2 Prerequisites

In this section, we consider some of the details around optimizing compilers more closely, as well as introduce several optimization strategies and a number of concepts that are foundational to this thesis.

2.1 Optimizing code

The concept of an "optimization" doesn't have very clear boundaries. Compilers are fundamentally just mapping source code to machine code, and even with well-defined language semantics, there are sure to be several natural ways to express any source code snippet in machine code. Necessarily, some are going to be a little faster than the others, and there may even be clever, nonobvious expressions that are much faster. While obvious at the extremes, the specific line between undertaking *optimization* as opposed to naively implementing the semantics is a subjective matter.

There are also several dimensions along which one may want to optimize, most notably execution time, binary size, and memory usage. In accordance with the No Free Lunch theorem, one is often sacrificed to improve another. To make matters even more complicated, it's not always obvious how effective an optimization is going to be because modern CPUs exhibit highly nonlinear behavior due to pipelining and caches. Saving a few pipelined instructions at the cost of a cache miss or branch mispredict is a big net loss.

Therefore, what constitutes an optimization and what one would consider to be counterproductive changes on a case-by-case basis.

With this in mind, we are going to look at several examples of transformations applicable to optimizing compilers but also hand-coded assembly, and consider the merits of either method. We are also going to see several common optimizations that are only practical in the context of automated optimization. At the end of this chapter, we will hopefully have seen the work that optimizing compilers do, the ways in which it is useful, how optimizations are intertwined with programming languages and the ISA, and what limitations there are.

2.1.1 Bit hacking

Avoiding multiplications is a very simple yet effective optimization. Compared to addition, the former often takes ~20 times as long[8:10], and so avoiding multiplications where possible is an easy way to squeeze some extra temporal performance out of programs. The general term for this kind of optimization (replacing a single operation with a faster equivalent) is *strength reduction*.

One way to avoid multiplying is resorting to bit shifts when one of the factors is close to a power of two. For example, `a * 2` can be more efficiently implemented as `a << 2`, or, somewhat less trivially, `a * 9` as `a + (a << 3)`. [6][8] In the real world, one might run into a code snippet such as the following.

```
int a = read_input();
int b = a * 9
```

Naively, the assignment to `b` might get compiled to the following RISC-V code:

```
addi    a1, zero, 9
mul     a0, a0, a1
```

In a world without optimizing compilers, if we wanted the performance gains afforded by the bit-shift trick, we would have to get used to writing `a + (a << 3)` instead of `a * 9`.

However, writing multiplications like this is illegible compared to the naive implementation. In fact, for certain optimizations, there might not even be a sensible way to express it in high level programming languages as we know them.

In such a situation, we would have to write everything directly in machine code, or hand-optimize the code using inline assembly. Even though not strictly necessary in the case of our example, consider an inline assembly version for purposes of illustration.

```
int a = read_input();
int b;

asm ("slli %1, %0, 3\n\t"
     "addw %1, %1, %0"
    : "=r" (a)
    : "r" (b));
```

Performance aside, this is very poor quality code.

- It is far removed from how programmers would expect to write a multiplication by 9.
- The semantics of the snippet (multiply `a` by 9) is hard to identify because we are using the bit shift trick *and* hiding it in an assembly string. Knowledge of the ISA is required to understand the code.
- Inline assembly is not necessarily a thing in other programming languages that we might want to use.
- This fairly verbose piece of code needs to be repeated in every place where we want maximum performance, and the same goes for all other optimizations.

It is obvious that any performance-optimized code like this would inevitably become an unmaintainable mess. In this world, code quality and performance are mutual exclusives.

Even worse, the most efficient form is always going to be machine-specific. So if multiple architectures need to be supported, we might end up with a monster like this:

```
int a = read_input();
int b;

#ifdef ARM
asm ("add %1, %0, %0, asl #3"
    : "=r" (a)
    : "r" (b));
#endif
```

```
#ifdef X86
asm ("mov %1, %0\n\t"
     "sal %1, 3\n\t"
     "add %1, %0"
    : "=r" (a)
    : "r" (b));
#endif

#ifdef RISCV
asm ("slli %1, %0, 3\n\t"
     "addw %1, %1, %0"
    : "=r" (a)
    : "r" (b));
#endif
```

Here, we are not only obfuscating the semantics, but also turning our code into a portability nightmare because platform support needs to be added explicitly and rewrites conducted for each platform individually. Obscure platform-dependent bugs might be introduced if subtle mistakes are made in the assembly strings.

In a similar vein, division is even slower than multiplication, and in some cases divisions can be avoided, instead using multiplications and/or shifts. As an example of such a transformation, with `-O0`, clang naively compiles the statement `return a / 3` to this x86 machine code:

```
mov     eax, dword ptr [rbp - 4]
mov     ecx, 3
cdq
idiv    ecx
pop     rbp
ret
```

With `-O3`, we instead get:

```
movsxd  rax, edi
imul    rax, rax, 1431655766
mov     rcx, rax
shr     rcx, 63
shr     rax, 32
add     eax, ecx
ret
```

Note in particular that the `idiv` instruction is completely gone. Thanks to the use of magic numbers and bit-hacking wizardry, we can avoid a slow `idiv` instruction, however the semantics of the operation is even more obfuscated, to the point where the disassembly is incomprehensible without explicit knowledge of the trick and the exact magic number.

2.1.2 Loop unrolling

Loop unrolling is another typical optimization. It saves a couple jump instructions and an almost guaranteed branch mispredict by transforming a loop with a fixed number of iterations

```
for (i = 0; i < 10; i++)
    sum = sum + a[i];
```

into a linear block of statements:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
// ...
sum = sum + a[9];
```

This example, too, illustrates the tradeoff between performance and legibility. The optimized version is far from how programmers would want to write (and read) that code.

Note that there is another tradeoff at play, trading space for time due to the increase in binary size, so for larger numbers of iterations this optimization is not necessarily worth it, especially considering the low cost of loop-related jumps on machines with even the most bare-bones branch prediction.

2.1.3 Inlining

Inlining is a very broadly applicable transformation; in fact, it can be employed anywhere a call is made, and will speed up execution of that call.

```
int abs(int n) {
    if (n < 0) return -n;
    return n;
}

int main() {
    int i, sum = 0;
    for (i = 0; i < 1000; i++) {
      int x = read_input();
      sum += abs(x);
    }
    return sum;
}
```

Calling the `abs` procedure this often is expensive: on each loop iteration, stack space needs to be allocated, parameters passed and the return address remembered; and right after the call everything needs to be undone again. We could skip all of the work done by the procedure prologue and epilogue by *inlining* the procedure:

```
int main() {
    int i, sum = 0;
    for (i = 0; i < 1000; i++) {
      int x = read_input();

      if (x < 0) sum -= x;
      else sum += x;
    }
```

```
    return sum;
}
```

Clearly, though, this is going to become tedious if the `abs` procedure is to be used in multiple places.

```
vec_4d vec_abs_4d(vec_4d* vec) {
    int w, x, y, z;

    if (vec->w < 0) w = -vec->w;
    else w = vec-> w;

    if (vec->x < 0) x = -vec->x;
    else x = vec-> x;

    // ...
}
```

Again, we're trading aspects of code quality such as legibility and maintainability for performance; this has been a common theme in the preceding examples, all of which are very common optimizations. Because these tradeoffs exist everywhere, deep-reaching manual optimization typically is simply infeasible when real-world issues are taken into account.

An optimizing compiler is a sensible solution to this problem. This is in addition to several advantages offered by optimizing compilers.

- No encyclopedic knowledge of optimization tricks is required
- Much faster speed of development
- Code has a chance to remain legible
- Ease when making changes
- Code remains ISA-agnostic

2.2 Compiler optimizations

We have covered a few optimizations in the previous section already; now we will consider a few more optimizations relevant specifically to optimizing compilers (and this thesis) and less common for manual optimization. Compilers typically apply optimizations in a series of passes each looking for its own set of specific improvements to be made.

Some compiler optimizations are best implemented closer to source code (AST) form while others are best implemented closer to binary form. This completely depends on the transformation.

For example, optimizations requiring some deeper information about the program which is difficult to represent or find out in a binary are clearly best applied close to the source code level. Other optimizations, however, might explicitly operate on individual instructions; and so only really make sense to apply when closer to binary form. Finally, some optimizations can be implemented in either case, though one might be more practical to implement.

2.2.1 Peephole optimization

Peephole optimizations are transformations where short sequences of compiled machine instructions are replaced with equivalent but faster versions. Therefore, peephole optimizations clearly belong to the type of optimization to be done close to binary form.

The name stems from the fact that we are considering sequences of instructions in isolation: the compiler's work here is akin to sliding a peephole over the machine code and looking for improvements without considering the broader context (unless necessary).

This kind of optimization can catch inefficient emitted code, such as subsequent redundant instructions, or ones that undo each other's effects. However, peephole optimizations can be quite sophisticated, too. As another example, re-consider the `return a / 3` division example given before. We can easily specify a pattern that matches the inefficient `idiv` version

```
mov     ecx, 3
cdq
idiv    ecx
```

and replace it with its the more efficient form

```
imul    rax, rax, 1431655766
mov     rcx, rax
shr     rcx, 63
shr     rax, 32
add     eax, ecx
```

So this rather non-straightforward optimization can easily be done by a peephole optimizer, too.

2.2.2 Superoptimization

Superoptimizers are a type of optimizer that conduct the exact same procedure as a peephole optimizer but aim to automate finding transformations.

The seminal paper [5] implemented a naive approach but still managed to get surprisingly good results. Given some code snippet to optimize, the approach was to exhaustively test all shorter code snippets; ones that produced the same result in several test cases were taken to be equivalent.

Using modern SMT solvers, actually *proving* equivalence for machine code snippets has become quite feasible, and therefore, so has the problem of automatically searching faster versions of machine code snippets which are proven to be equivalent.

Superoptimizers do tend to find interesting transformations that aren't obvious, and modern implementations which use SMT solvers [7] are in fact in production use for example in LLVM.

2.2.3 Dead code elimination

A perhaps surprisingly common theme in code is that often there are portions of the code which can't be reached. Sometimes unreachable code is obvious:

```
printDebug = false;

// ...

if (printDebug) {
  printf("vector: %s", vec);
}
```

In cases such as this, code is unreachable on purpose to avoid always printing debugging output. However, often there are non-intentional, less obvious cases, for example in contrived if-else statements or due to complex conditions which will always evaluate to `false` anyway.

Such code is called **dead code** and poses a nice optimization target. Code that's never used can be safely removed which yields a smaller binary. One way to find such code is by traversing the control flow graph, a construct that we are going to consider in more detail in a later section.

2.3 Caveats

Sadly, optimizers aren't almighty.

While this quickly becomes obvious in practice, it can also be formally shown. A fully optimizing compiler (i.e. one that would transform any program to its most optimal form) would solve the halting problem:

- For any program that never halts and produces no output, the shortest form is the one-line infinite loop `while(true);`
- To see whether there is some input for which some program P halts, feed P to a fully optimizing compiler and check if the output is `while(true);`

On a positive note, this result is often referred to as the *full employment theorem for compiler writers*.

2.3.1 Language features and optimization potential

Algorithmic improvements are one type of optimization that compilers generally do not implement. For example, if some code is using insertionsort on large lists, it would be nice if the compiler could replace it with quicksort.

In most widely used programming languages, such big-picture optimizations cannot be done by compilers, given that automatically determining the semantics behind a nontrivial code snippet and proving that we are going to replace it with an equivalent but better one is hard to impossible, depending on the situation.

One reason for this is a lot of context is required to make an informed decision. For small lists, insertionsort is actually much faster than quicksort. How could it be known to a compiler which one is appropriate?

Secondly, the presence of certain language features such as reflection or `eval` statements, as well as a *lack* of other features such as a way to specify pre- and postconditions and invariants, may also be a hindrance in implementing such optimizations. Depending on the feature set, certain code properties may be impossible to prove in reasonable time. Therefore, while macro-scale optimization can be done successfully, compiler optimizations typically happen on a micro scale in practice.

Dead code elimination nicely illustrates the tradeoff between language features and optimization potential. In object oriented languages, entire classes might be "dead":

```
import java.lang.*;

class Unused {
  public String toString() { return "Unreachable"; }
}

class Main {
  public static void main(String[] args) throws Exception {
    System.out.println("Hello World!");
  }
}
```

We might be enticed to mark all of the `Unused` class as dead code and just skip compiling it. But proving that there is no direct way for control flow to move into some code block isn't necessarily enough. If jump addresses are determined at runtime, for example via reflection, then we cannot just remove the class `Unused`.

```
import java.lang.*;

class Unused {
  public String toString() { return "Unreachable"; }
}

class Main {
  public static void main(String[] args) throws Exception {
    System.out.println(Class.forName(args[1]).newInstance());
  }
}
```

In this case no direct reference to the `Unused` class is ever made, however its `toString` method may still wind up getting called. We cannot remove the class due to the reflection feature of Java. We'd trigger a `ClassNotFoundException` when the class was in fact properly defined in source code - a clear violation of the semantics.

One last hope might be the idea that proving the absence of `Class.forName` calls may allow us to get rid of the unused code after all. This however is also not possible because of Java's dynamic linkage. We might not run into a `Class.forName` call during compilation, but when the bytecode is actually run an entirely different implementation of some class might be in place.

Therefore, this potential optimization cannot be done due to some of the features afforded to programmers by Java.

Anoter notable, rather subtle language feature with an impact on the optimization potential, is pointer aliasing. Pointer aliasing used to provide Fortran with an edge over C when it comes to potential for compiler optimizations.

In C, it is assumed that any pointer may point to the same address as some other pointer. Fortran does not make this assumption. This assumption may not seem like it has far reaching implications, but it is the reason C reads from memory much more often than Fortran in some situations. Consider as an example:

```
void vec_extend(vec_2d* source, vec* dest) {
  int i;
  for (i = 0; i < dest->n; i += 2) {
    dest[i]     = source[0];
    dest[i + 1] = source[1];
  }
}
```

Here, a worthwhile optimization would be caching the two `source` values in registers so they aren't read from memory with each assignment. However, due to pointer aliasing, C does not assume that writing to `dest` won't overwrite `source`, and so it has to read both values in every single loop iteration. Fortran does not do this and so the values can be cached.

With C99 parity has been restored by introducing the `restrict` keyword. When applied to a pointer, it acts as a "promise" that this pointer is not going to overlap with any other pointer's memory region. Therefore, the same optimization can in fact be done by a C compiler by adding `restrict`:

```
void vec_extend(vec_2d* restrict source, vec* restrict dest);
```

At this point we have seen concrete examples of how advanced language features such as reflection may hinder code optimization.

Certain other language features that allow compilers to deduce more information about programs exist, too, which in turn enables more powerful optimizations. However, contrary to the former, this latter kind of language feature is less common in widely used languages. It seems that language features desired by compiler writers are not always aligned with those desired by developers.

2.3.2 Runtime information

Typical compilers don't have access to any runtime information. As we have just seen, this limits the optimizations available to the compiler - for example, if jump addresses are runtime information as may be the case in a language with reflection.

Runtime information is however often available to the programmer. Therefore, that is also where the responsibility for implementing optimizations dependent on runtime information lies. Naive high-level code is likely to leave powerful optimizations on the table, even if compiled with `-O3`. The programmer must

still decide on questions such as the proper algorithm for the expected input size.

This issue is alleviated somewhat by JIT compilers, which only compile code when it is actually run (hence the name *just-in-time*). A JIT compiler, therefore, operates at runtime, may collect runtime information and undertake corresponding optimizations.

For example, TruffleRuby will dynamically choose the fastest algorithm for the given input size when the standard library `sort` function is called. Another widely used technique is profiling the code as it runs and spending more effort on optimizing *hotspots* that run over and over again, such as inner loops.

JIT compilers nicely quantify the power of runtime information. In a synthetic benchmark, LuaJIT achieves a 25% reduction in execution time compared to C[1], even though conventional wisdom says interpreted languages such as Lua are a lot slower than compiled languages, of which C is one of the fastest.

2.3.3 Compiler bugs

Another caveat is that the complexity of optimizations isn't removed, but merely hidden away in the compiler. This is a big improvement in that optimizations only have to be written once in one place, however, it adds spatial and temporal costs to compilation.

Furthermore, compiler engineers aren't infallible, and so bugs do sneak in. GCC's bugtracker has handled just over 100,000 bugs[2] at the time of writing, and while compiler bugs aren't typically an issue, they can produce very obscure issues if they do occur.

2.3.4 Tradeoffs

Finally, automatic application of optimizations yields many benefits, however, as noted in the unrolling example, some optimizations come with fairly situation-dependent tradeoffs and the compiler may not always make the optimal choice as to whether a transformation is worth it in the given circumstances.

The introduction of optimization level flags alleviates this issue somewhat by giving the programmer more control over the employed optimizations. Programmers can choose optimization levels such as:

- `-O0` through `-O3` which will apply increasingly aggressive optimizations, where `-O3` is the first level which will employ transformations that trade memory for time.
- `-Ospace` which does the inverse, trading time for memory usage and binary size
- `-Ofast`, which implements all optimizations in `-O3` and also drops compliance with certain parts of the language specification to get even more performance.

To gain even more control, different portions of the code can be compiled with different flags, depending on what is most appropriate.

[1] https://gist.github.com/spion/3049314
[2] https://gcc.gnu.org/bugzilla/buglist.cgi?bug_status=___all___&query_format=specific

2.3.5 Power of manual optimization

A motivated individual can outdo even the most advanced compiler. In 2003, GotoBLAS was the state-of-the-art BLAS, hand-written by Kazushige Goto in x86 assembly during a sabbatical. BLAS (meaning Basic Linear Algebra Subprograms) provides a set of elementary routines to be used for solving problems in linear algebra. Due to the sheer number of calls to BLAS, e.g. when solving a large matrix, its performance is critical. GotoBLAS boosted the performance of a supercomputer cluster from 1.5TFlops to 2TFlops just by replacing the previously used BLAS [4].

However, the cost in portability is also nicely illustrated as GotoBLAS is optimized not just for a specific ISA or even a specific CPU vendor but for a specific microarchitecture. Optimal performance is reached only on the targeted Intel cores. The final 2008 release of GotoBLAS is optimized for Nehalem, the architecture used in the initial release of Intel's Core i series, for example the Core i7-960.

2.4 Binary optimizers

A *binary optimizer* is a standalone tool that takes binaries as input; that is, it isn't part of any specific compiler and instead performs static analysis on its input binary to find possible optimizations. Such optimizers are commonly called *object code optimizer* or *binary optimizer*. The main result of this thesis is a simple binary optimizer.

The design choice to create a binary optimizer has various implications. The obvious benefit is that it works on any binary even if no source code is available. One scenario where this can be useful is when working with legacy code where the source code has been lost. Another common scenario is in dealing with proprietary software.

With *IBM Automatic Binary Optimizer for z/OS*[3], there exists a commercial binary optimizer that may cover a lot of cases where the source is inaccessible. It works with COBOL binaries, a very old language that is however still often relied on in business; one could imagine that many ancient source files have not survived the test of time.

As another benefit, the compiler remains simple. In modern optimizing compilers, there are tens of thousands of lines dedicated to specifying peephole optimizations alone; for example, LLVM has some 30,000 lines of C++ dedicated to this[4]. Other compilers such as GCC have created a domain-specific language to simplify the task.

On the other hand, there are certain limitations:

- Before any optimizations can be done, the binary needs to be inspected for potential optimization sites.
- When applying optimizations, we need to make sure that the binary does not break in unexpected ways.

[3] https://www.ibm.com/products/automatic-binary-optimizer-zos
[4] https://github.com/llvm/llvm-project/tree/main/llvm/lib/Transforms/InstCombine

- We are also limited to expressing optimizations in terms of a pattern and its replacement.
- Finally, certain transformations are more difficult to detect and apply on the binary level as compared to an optimizer that operates on some higher-level intermediate representation.

We will now consider these points in more detail.

2.4.1 Static analysis

In general, while an optimizing compiler necessarily already has an abstract representation of the program due to the process of compilation, a binary optimizer needs to construct such a representation from the binary code. This representation must then be analyzed in order to apply optimizations. Meanwhile, the process of compilation automatically yields a decent wealth of information about the program, so an optimizing compiler can skip some of the more basic analysis. Clearly, there is no need to look for constructs such as loops if we know exactly where we emitted loop code.

The process of analyzing binaries is called *static program analysis*. (*Dynamic* program analysis is performed on running programs.) Static analysis is an abstract term, covering methods that range from simple to extremely convoluted, and technically it is also often done manually by humans, in a process called *code review*.

In the context of this thesis, we created a simple static analyzer. This static analyzer is subject of [9]. A rough overview of the analyzer follows in a later section.

2.4.2 Subtle bugs

In a previous section, we discussed peephole optimizers, which are fundamentally just executing search-and-replace operations on patterns of instructions. Naturally we might be enticed to just replace all occurences of a slower sequence of instructions with a faster version.

However, we cannot be quite that trigger happy. If there is a jump instruction somewhere in the code which jumps into the middle of the sequence of instructions that we are looking to replace, then the new program is practically guaranteed to be broken. This is one of the issues that need to be taken into account when conducting static analysis.

Of course, the problem posed by subtle bugs isn't unique to binary optimizers, and as mentioned before, bugs remain an issue in complex production compilers.

2.4.3 Performing optimizations

Performing optimizations may be a bit more involved when working on a binary than working on an intermediate representation designed with that purpose in mind. As mentioned before, this depends on the kind of optimizations to be performed - some optimizations are simpler in AST form, others are simpler in binary form.

As an example of the former kind, there's nothing preventing loop optimizations from being performed on binaries, though closer to source code form, they are easier to detect and perform [1:7]. An example of a loop transformation is *hoisting*:

```
for (i = 0; i < n; i++) {
  int k = 0;
  // ...
}
```

Here, C semantics will tell the machine to de- and reallocate a local variable with every single iteration of the loop. This is obviously wasteful. To avoid this, we can just allocate it once outside of the loop.

```
int k;
for (i = 0; i < n; i++) {
  k = 0;
  // ....
}
```

In a binary optimizer, we need to invest a lot of effort to correctly identify loops in the first place. An optimizing compiler gets this knowledge "for free" when the parser sees a `for` keyword.

2.5 selfie

Our optimizer is built around the selfie ecosystem. Selfie is an ongoing project of the Computational Systems Group at University of Salzburg.

> "Selfie is a self-contained 64-bit, 11-KLOC C implementation of:
>
> 1. a self-compiling compiler called starc that compiles a tiny but still fast subset of C called C Star (C*) to a tiny and easy-to-teach subset of RISC-V called RISC-U,
> 2. a self-executing emulator called mipster that executes RISC-U code including itself when compiled with starc,
> 3. a self-hosting hypervisor called hypster that provides RISC-U virtual machines that can host all of selfie, that is, starc, mipster, and hypster itself, and
> 4. a tiny C* library called libcstar utilized by selfie." [3]

The optimizer is itself written in C*, the full specification of which is given in [2]. A quick tour of C* would have to mention that it features only two types, an unsigned 64-bit integer (`uint64_t`) and a pointer to an unsigned 64-bit integer (`uint64_t*`). Consequently, there are neither structs nor arrays, not to mention any more complex constructs.

The elementary control flow and arithmetic operations are present, as well as the modulo operator, however bitwise and Boolean operations as well as complex control flow statements such as `switch`es are excluded. Procedures however are fully supported.

Therefore, a lot of footwork must be done manually, for example arrays and structs have to be emulated using pointer arithmetic. The bare-bones language

obviously adds to the difficulty of writing an optimizing compiler.

However, this minimalism allows students to quickly pick up selfie in classes and work on extending it. Thus, avoiding complexity in the compiler is one reason why we chose to build a binary optimizer. Implementing optimizations directly in the selfie compiler would have made it much more difficult to understand and work on.

2.6 Conclusion

We have seen that compilers wield a massive arsenal of optimizations that would be difficult to even come close to as an individual programmer, and that all of this arsenal is automatically applied on every compilation at the modest cost of additional resources consumed at compile time.

Offloading optimization to the compiler incurs no additional cost in readability, maintainability, and portability, which would just not be the case with manually performed optimizations.

However, due to the difficulty of program analysis, compilers lack a view of the big picture - it is up to the programmer to realize performance gains from algorithmic improvements. Furthermore, if optimal performance is an absolute requirement, there still is no alternative to handcrafted assembly.

In short, optimizing compilers offer a very compelling package for developers who do not want to spend most of their time carefully optimizing the performance of their programs through measures such as inlining, though care is still needed in making big-picture decisions such as choosing the fastest algorithms.

3 Design

We implemented a binary optimizer for RISC-V binaries. The optimizer's main components are a static analyzer, an optimizer capable of applying a series of transformations, and a collection of such optimizing transformations. The static analyzer runs first, constructing a control flow graph and collecting information about the known machine state at each instruction.

This information is fed to the optimizer which runs next. The optimizer implements several peephole optimizations, dead code elimination, and it also removes instructions that can be proven to have no effect on the program output.

The output from the optimizer requires some fixing up as some of the transformations break jump addresses. This is done in a separate final stage.

In the next sections, we will consider the design and tasks of each of these components in more detail, while the next chapter is going to give implementation details.

3.1 Static analyzer

The static analyzer performs the task of collecting as much useful information as possible for the optimizer. In our work, analysis is done by iterating over the code segment of the input binary, collecting information about the changes in the machine state effected by each instruction.

For each instruction, there is a memory structure that contains information about the machine state that is guaranteed to be true in every possible execution of that instruction.

An *execution* of the program refers to running the program once with any specific input. If we say X is true for any possible execution of the program, this means that it is impossible for X to ever not be true, no matter what inputs you feed the binary.

A straightforward example may be the following claim: the register `t0` is always 0 just after executing the third instruction of the program. This may simply be the case if the third instruction sets `t0` to 0, but we will see that things can quickly get convoluted.

Information collected by the static analyzer includes the *control flow graph*, *dataflow information*, and *liveness information*.

3.1.1 Control flow graph

A control flow graph is a graph representation of a program. Let a *jump* be any change to the program counter that is not just a trivial increment to the next instruction. Each node represents a sequence of statements that neither contains a jump nor is the target of one. We may call such a sequence a *block*. Jumps, then, sit in between blocks, and each one represents a directed edge from the source block (*out-edge*) to the target block (*in-edge*).

Clearly, only statements that manipulate the control flow are of relevance to the control flow graph. What the code actually does is irrelevant. Control flow

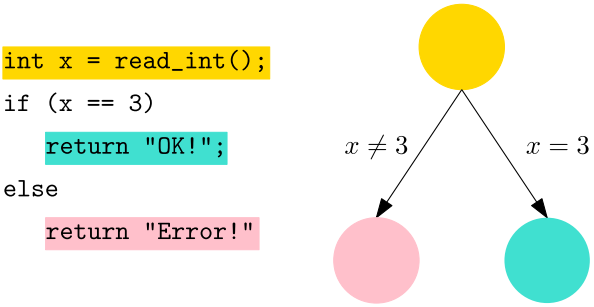

Figure 1: A simple code snippet and corresponding CFG

graphs may be constructed on both source code or binary code, and there may be loops, but no multi-edges.

Control flow graphs are relevant to optimization because they let us figure out whether we can apply an optimization without breaking the semantics of the program. Suppose that in the process of analyzing a binary, we run into the following, familiar assembly:

```
0x08    addi    a1, zero, 9
0x0C    mul     a0, a0, a1
```

We know that `a0 + (a0 << 3)` is a faster way to compute the result of the multiplication `a0 * 9`, and so we might be enticed to make the replacement:

```
0x08    slli    a1, a0, 3
0x0C    add     a0, a0, a1
```

Assume that later there is a jump which goes to the `mul` instruction we were going to replace.

```
0x24    addi    a1, zero, 12
0x28    jal     ra,-7[0x0C]
```

If we did this replacement, any execution that took the jump at `0x28` would now find an `add` in place of the original `mul`. The semantics would change from `a0 * 12` to `a0 + 12`.

The CFG lets us avoid all situations like this at once. The rule is that we may only replace sequences with neither an in- nor an out-edge, that is, only a single vertex in the CFG at a time.

In our example, the sequence that we were going to replace will decompose into more than one vertex in the CFG. Therefore, we know the replacement is not going to work.

The CFG is also of use for dataflow analysis, the next phase of program analysis.

3.1.2 Dataflow analysis

Dataflow analysis is a general term for collecting information about what may happen to the machine state at runtime. In our case, we are exclusively looking for information that is going to be true universally across all possible executions,

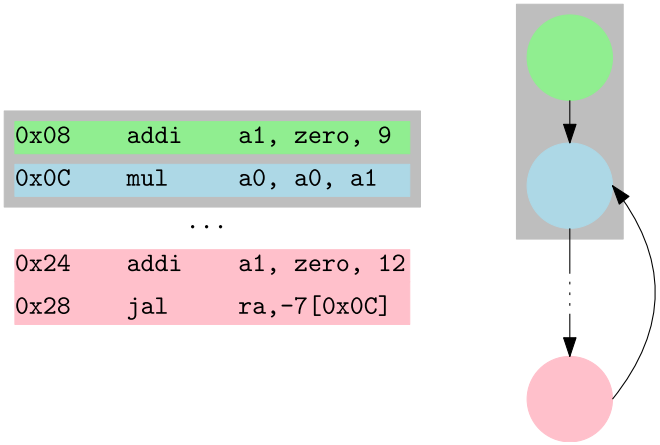

Figure 2: Decomposition of the sequence (in grey) into two vertices due to the in-edge

therefore, the data we obtain is fairly conservative. In contrast to constructing the CFG, the semantics of the instructions are now central to the analysis.

There are many kinds of information that one might attempt to collect about a given binary. For our purposes, we limited the dataflow analysis to keeping track of the values of the registers at any point on the program, and the *liveness* thereof (expanded on in the next section).

Our analyzer allocates a block of 32 words for each instruction in the binary. Each word corresponds to a register. We then iterate over the control flow graph. If, at the i-th instruction, a register can be proven to always have the same value at that point in every execution, this value is saved in the 32-word block belonging to that instruction.

This may sound like an uncommon scenario, but really it is not. Consider the previous example.

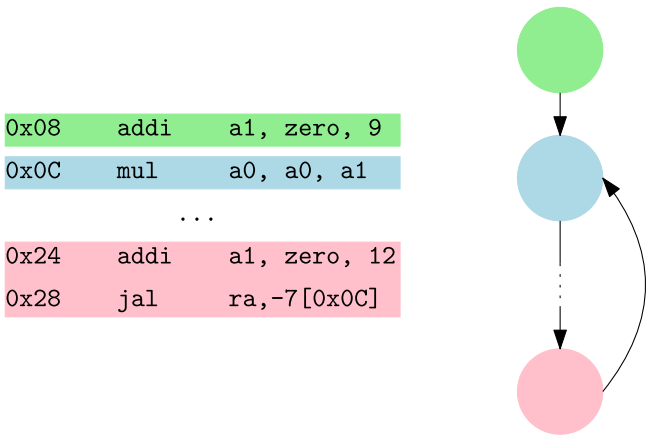

Figure 3: Example program and CFG

In the pink block, we have an `addi a1, zero, 12` at `0x24`. Already, this means `a1` is always going to hold the value 12 after executing the instruction at `0x24`. The next instruction does not modify that value, therefore `a1` is still going to be 12 at `0x28`. We say that the definition of `a1` at `0x24` *reaches* `0x28`.

Similarly, at `0x08`, `a1` is defined to be 9. However, it is not necessarily the case that `a1` is still holding that value at `0x0C`, since the definition `0x08` is not the only way to reach `0x0C`. We may also reach `0x0C` via the jump instruction at `0x28`, and `0x28` in turn is reached by the definition `0x24`. In other words, `a1` may be either 9 or 12.

In such a situation, we simply treat `a1` as an *unknown* value, which effectively just means no optimizations that require a known value of `a1` are conducted.

For clarity, the following table reproduces the values of `a1` that our analyzer would collect.

Instruction	Value of `a1`
`0x08`	9
`0x0C`	unknown
...	...
`0x24`	12
`0x28`	12

Clearly, the CFG is an indispensable tool for dataflow analysis. Knowledge of in- and out-edges is required to correctly handle the peculiarities introduced by jumps. Though hardly difficult to imagine, exactly how this dataflow analysis is useful in conducting optimizations will be considered in detail in a later section.

We have also made attempts to track the contents of memory in the same way. Since the addresses are only known at runtime, we cannot track the state of dynamically allocated memory. However, global variables could in theory be analyzed in the same way registers are analyzed, and would provide a wealth of additional information. Sadly, the analyzer would have to have gotten decidedly more complex in order to support this.

3.1.3 Liveness information

Liveness information pertains to registers and variables. A register is *live* if its current value is going to be read in the future, and *dead* if the opposite is true.

Clearly, if a register is dead, any computation on it is wasted cycles since the value is going to be overwritten before it will be read. Instead, we may remove that computation and use the now freed register as a temporary in the meanwhile.

To illustrate this concept, consider the following sequence of instructions.

```
0x08    addi    a1, zero, 9
0x0C    mul     a0, a0, a1
0x10    ld      a1, 0(sp)
0x14    addi    a0, zero, 3
```

We can see that after computing the multiplication `a0 = a0 * a1`, there comes a load, and finally `a0` is defined to be 3. Therefore, the result of the multiplication is simply overwritten. We might as well skip it.

```
0x08    addi    a1, zero, 9
0x0C    ld      a1, 0(sp)
0x10    addi    a0, zero, 3
```

One might have astutely observed that by removing the multiplication, the initial definition `a1 = 9` has also become dead. It is simply overwritten by the load instruction that comes next. Therefore, removing dead instructions may *cascade* - it may be necessary to iterate a few times before we reach a fixed point. For our example, that fixed point is the following.

```
0x08    ld      a1, 0(sp)
0x0C    addi    a0, zero, 3
```

When considering liveness in general, the thing to look out for are jumps in between definitions. If there are out-edges, the value must be overwritten before its next use on all paths. In-edges are not relevant to liveness.

More concisely, we might say that to be dead, a definition must reach another definition before it is read, on all possible paths. This is trivially true in our example since there is no jump, however in the real world the CFG is again a useful tool for handling these situations.

3.2 Optimization passes

Using the output from the analyzer, the optimizer performs a series of transformations in order to improve the spatial and temporal performance of the code.

The optimizer has a couple of peephole optimizations available, as well as dead code elimination and removing instructions that can be proven to have no effect on the machine state. We call the latter kind *effective no-ops*, or `enop` for short.

Transformations are applied separately, for each one the binary is iterated over once, looking for applicable sites. The order of the optimization passes does, in fact, matter. There might be sites in the binary where two transformations can be applied. Depending on the transformations in question, it may be the case that one is universally better than the other, however typically it is difficult to tell, especially given that there might be cascading effects.

As mentioned before, these are often difficult tradeoffs, and production optimizers will typically employ heuristics to come to a solution in reasonable time. We have taken a similar approach by testing several orderings with selfie's self compilation and choosing the one with the largest number of instructions removed.

The only exception to this are `enop`s, which are always identified and removed in the same pass as dead code. This has two reasons:

- `enop`s are not fundamentally different from dead code, and can be found with almost the same algorithm.
- Should an `enop` and dead code ever overlap, the only sensible course of action is removing the instruction in question anyway.

In short, order does not matter, and conducting both transformations in the same pass is more efficient anyway.

3.3 Fixup

When removing instructions, jump addresses will break and will need to be fixed. Consider the following example.

```
0x4930: 0x0180006F: jal zero,6[0x4948]
0x4934: 0x01043283: ld t0,16(s0)
0x4938: 0xFF810113: addi sp,sp,-8
0x493C: 0x00513023: sd t0,0(sp)
0x4940: 0xC39FF0EF: nop
0x4944: 0x00000513: addi a0,zero,0
0x4948: 0xFB01B283: ld t0,-80(gp)
```

If we were to simply remove the `nop`, then the initial `jal` is now pointing past the end of this sequence:

```
0x4930: 0x0180006F: jal zero,6[0x4948]
0x4934: 0x01043283: ld t0,16(s0)
0x4938: 0xFF810113: addi sp,sp,-8
0x493C: 0x00513023: sd t0,0(sp)
0x4940: 0x00000513: addi a0,zero,0
0x4944: 0xFB01B283: ld t0,-80(gp)
```

(Of course, the same thing also holds for every other jump offset that includes this `nop`.)

So the proper course of action has to include fixing the jump offset so that it is still pointing to the same instruction after removing the `nop`:

```
0x4930: 0x0180006F: jal zero,5[0x4944]
0x4934: 0x01043283: ld t0,16(s0)
0x4938: 0xFF810113: addi sp,sp,-8
0x493C: 0x00513023: sd t0,0(sp)
0x4940: 0x00000513: addi a0,zero,0
0x4944: 0xFB01B283: ld t0,-80(gp)
```

In our optimizer, this fixing up is done separately from the optimization passes. Initially, instructions that are to be removed are just replaced with a `nop` as a means of tagging them for later removal. After the optimization passes complete, a separate tool (`noprm`) does the task of removing `nop`s and fixing the jump addresses.

4 Implementation

The simplicity of the selfie compiler gives us a lot of optimization potential to work on. Most of the optimizations, therefore, are specifically tailored towards RISC-U code emitted by selfie, although they work on any RISC-V binary thanks to the fact that RISC-U is a subset of RISC-V.

But obviously, the transformations will be limited to the scope of RISC-U, that is, operating on no more than 14 instructions. As an example of the limitations imposed by this, note that RISC-U lacks bit shift instructions. This means all of the bit hacking optimizations mentioned at the start of this thesis cannot be implemented.

The next section is going to give an overview of the implementation of the optimizer. The implementation of the analyzer is covered in Thomas' thesis; here, only a short description of the API will follow.

4.1 Analyzer

The control flow graph, dataflow information, and liveness information is stored in global data structures, and accessible via accessor methods.

The CFG data structure is implemented as an array of linked lists. A global variable holds a pointer to a contiguous block of memory, where the i-th word points to the list of edges of the i-th block. An edge is simply two dynamically allocated machine words: the target block of that edge, and a `next` pointer.

Furthermore, it should be noted that our CFG implementation interprets each instruction as its own block; in effect this means that when replacing a sequence of instructions, instead of checking that it is a single block, we check that there are no in-edges.

The CFG API allows several operations, mainly used by the analyzer. The portion of the API that is of interest to our optimizer is simply the following method:

```
uint64_t is_return_target(uint64_t pc) {
  return *(return_targets + pc/INSTRUCTIONSIZE);
}
```

For dataflow analysis, the API is the following:

```
// Array of machine states, one for each instruction.
uint64_t *machine_states = (uint64_t *) 0;

// Check if `reg` has a known value in every execution
// at the instruction that `state` belongs to.
uint64_t is_reg_known(uint64_t *state, uint64_t reg);

// Get the value of a register if known
uint64_t get_reg(uint64_t *state, uint64_t reg);
```

Similarly to the results of our dataflow analysis, liveness information is stored per-instrution in a block of words. The optimizer consults this information

mainly via one single method:

```
// 1 iff `reg` is live in every execution
// at the instruction that `livedead` belongs to.
uint64_t is_reg_live(uint64_t* livedead, uint64_t reg);
```

4.2 Optimizer

The optimizer has two fundamental kinds of transformations available: identifying and removing useless instructions, and applying predefined peephole optimizations. Therefore, the core API of the optimizer is the following two methods:

```
// Wherever possible, apply the peephole optimization
// with the ID `pattern`
void patch_peephole(uint64_t pattern);

// Find and remove useless instructions
void patch_enops();
```

4.2.1 Peephole optimizer

The peephole optimizer has fairly general capabilities; while there is no DSL, adding new optimizations is still only a matter of defining a few variables and functions. For example, the following code defines a pattern that matches a `jal zero,1` instruction.

```
// Define an ID
uint64_t PATTERN_JAL_NOP = 1;

// Define the pattern to be matched
void pattern_jal_nop(uint64_t matched_instructions) {
  number_of_instructions_in_pattern = 1;

  // 4194415 = 0x0040006F = jal zero,1
  match_ir = 4194415;
}

// Register the pattern in the update function
void update_pattern(uint64_t matched_instructions) {
  reset_pattern_matcher();

  if (current_pattern == PATTERN_JAL_NOP)
    pattern_jal_nop(matched_instructions);
  // more patterns may follow here...
}
```

Using the rest of the optimizer as a library, this is already all of the code required in order to create a bare-bones, but fully operational optimizer:

```
int main(int argc, char** argv) {
  setup(argc, argv);
```

```
  patch_peephole(PATTERN_JAL_NOP);

  selfie_output(binary_name);
}
```

The pattern matching engine also implements wildcards. For example, one can match `addi` instructions with `t0` as the source or target register, or any of the temporaries `t0`-`t6`, or simply any register. Similar flexibility is afforded for the immediate values, either a fixed immediate value or any immediate value can be matched. The matched value can be referred to in the replacement pattern. This allows us to express more powerful transformations, we will see examples thereof when we take a closer look at the catalog of transformations in chapter 5.

The core method `patch_peephole()` boils down to iterating over sites that match the pattern and applying the transformation. To find matching sites, the engine must keep track of how many instructions in the pattern have already matched, and return its current position in the binary if the final instruction in the pattern matches.

This pattern matching procedure is fundamentally the same as substring search. We simply employ the naive worst-case $O(nm)$ algorithm, because peephole optimizations are generally very short sequences and so the theoretical worst case is not going to be a problem in practice.

The legwork of figuring out whether the current instruction matches the pattern is done by the method `instruction_matches()`. It contains a series of checks of the following form:

```
if (match_rs1 != ANY) {
  if (match_rs1 == ANY_TEMPORARY) {
    if (!is_temporary_register(rs1))
      return 0;
  } else if (match_rs1 != rs1)
    return 0;
}
```

This returns 0 if `rs1` doesn't match the specification of the pattern. The same is done for `rs2`, `rd`, `imm`, `funct3`, `funct7`, or the precise value of the instruction register, if specified by the pattern. If none of the checks return 0 early, the method returns 1 at the end.

Finally, if a match is found and a callback function for modifying the matched sequence is defined by the pattern, then that function is called. If none is defined, the matched sequence is overwritten with `nop`s, for later removal in the fixup phase.

4.2.2 Removing instructions

The method `patch_enops()` iterates over the binary and removes `enop`s and dead code:

```
void patch_enops() {
  uint64_t last_enop;
  last_enop = find_next_enop(0);
```

```
  while (last_enop != -1) {
    insert_nop(last_enop);
    last_enop = find_next_enop(last_enop);
  }
}
```

Most of the work is offloaded to `find_next_enop(uint64_t pc)`. This method returns the next `enop` after `pc`. The core of this method is the following condition:

```
if (apply_effects(state) == 1)
  if (test_states_equal(get_state(i), state))
    return i;
```

In other words, the current `pc` (`i`) is returned if, after applying the current instruction to a copy of the current state, the modified copy and the original state are still the same.

The key to making this work lies in what "applying the current instruction" is taken to mean. `apply_effects` reprocudes the semantics of all 14 RISC-U instructions, that is, if the current instruction is an `addi t0,zero,4`, then `apply_effects` will modify `t0` in `state` to be 4, and return 1.

However, if the current instruction is `add t0,t0,t1`, the situation isn't quite as clear. We can only proceed if we know the value of both `t0` and `t1`. If, for either of them, the analyzer couldn't identify a value that is going to hold in every execution (`is_reg_known()` returns `false`), then `apply_effects()` must return 0. There is no sensible way to apply the unknown effects of the current instruction to the known `state`.

With this code, it is possible to simply iterate over all hits returned by `find_next_enop()`, and mark all these instructions for removal. This will catch `enops` as well as unreachable code.

4.3 Fixup

After successful completion of the optimization passes, the binary should contain large numbers of `nop`s. Removing them is the final step to getting an optimized binary out of our optimizer, however care needs to be taken with jump addresses as explained in the previous section on fixup.

Fixup is implemented in a separate binary called `noprm`. It iterates over the binary twice. In the first pass, all of the `nop`s are removed, and in parallel, an array A is constructed, where $A[i]$ gives the new address of the i-th instruction. This array is then used in the second pass to fix the jump offsets in the binary. As can be seen easily, only $O(n)$ runtime and memory with small constants is required.

5 Transformations

The catalog of transformations employed by the optimizer was compiled via manual inspection of disassembled code emitted by selfie. Repeated patterns with potential for optimizations were noted and finally the most promising subset of the patterns was implented in the optimizer.

We will now consider each transformation in more detail.

5.1 Effective noops

An effective noop is any instruction that has no effect on the machine state. This isn't necessarily only the case for `nop`; other instructions may have no effect as well.

As a real-world example, consider this snippet from selfie's source code:

```
void reset_register_access_counters() {
  reads_per_register  = zalloc(NUMBEROFREGISTERS * REGISTERSIZE);
  writes_per_register = zalloc(NUMBEROFREGISTERS * REGISTERSIZE);

  *(writes_per_register + REG_SP) = 1;
  *(writes_per_register + REG_S0) = 1;

  // ...
}
```

The two latter assignments translate to the following assembly:

```
0x18EC(~1435): 0xFFFFF2B7: lui t0,0xFFFFF
0x18F0(~1435): 0x005182B3: add t0,gp,t0
0x18F4(~1435): 0x3C82B283: ld t0,968(t0)
0x18F8(~1435): 0xAA01B303: ld t1,-1376(gp)
0x18FC(~1435): 0x00800393: addi t2,zero,8
0x1900(~1435): 0x02730333: mul t1,t1,t2
0x1904(~1435): 0x006282B3: add t0,t0,t1
0x1908(~1435): 0x00100313: addi t1,zero,1
0x190C(~1435): 0x0062B023: sd t1,0(t0)
0x1910(~1436): 0xFFFFF2B7: lui t0,0xFFFFF
0x1914(~1436): 0x005182B3: add t0,gp,t0
0x1918(~1436): 0x3C82B283: ld t0,968(t0)
0x191C(~1436): 0xA701B303: ld t1,-1424(gp)
0x1920(~1436): 0x00800393: addi t2,zero,8
0x1924(~1436): 0x02730333: mul t1,t1,t2
0x1928(~1436): 0x006282B3: add t0,t0,t1
0x192C(~1436): 0x00100313: addi t1,zero,1
0x1930(~1436): 0x0062B023: sd t1,0(t0)
```

Notice that the value 8 is loaded into `t2` at `pc=0x18FC` (call this instruction a), and again at $b =$ `0x1920`. The second write at b is simply repeating something we already did before, so it has no effect on the machine state - it is effectively a no-op.

Before we can declare this instruction as redundant, get rid of it and call it a day, we need to make sure two conditions hold, otherwise we are going to run into runtime crashes.

The first condition is simply that no other value is written to `t2` in between a and b. Of course, if that were the case, then b is no longer redundant.

The second condition brings us back to the control flow graph. If there is an in-edge in between a and b, there are two possible cases:

1. `t2` is guaranteed to have the value 8 in every possible execution of the path along the in-edge, or
2. `t2` is not guaranteed to have the value 8.

Fortunately, the analyzer provides us with enough data to handle everything elegantly. Recall that for every instruction, it maintains a list of register/value pairs which it can prove to always have the same fixed value in every possible execution.

More specifically, the analyzer reimplements the semantics of most of the 14 RISC-U instructions. If the previous machine state contains enough knowledge to find the result of the current instruction, then that result is computed and stored for the current machine state.

For example, if we encounter an `addi t0,t1,t2` at the i-th instruction, and we know that `t1=4` and `t2=8`, then the analyzer concludes that `t0=12` at the $i+1$-th instruction, unless there is an in-edge of course.

This list of known machine states can be used to handle both conditions at once. All the work that's left for the optimizer to do is to check whether the current machine state is equal to the next one. If so, the instruction in question does nothing in every possible execution of the program and can safely be removed.

Returning to our example, there is no in-edge in between a and b, and the definition `t2=8` at a reaches b. This is taken into account by the analyzer, and therefore, the known machine states just before and just after execution of b are equal. The optimizer sees this and so it knows it can safely remove b. This wraps up finding and eliminating `enop`s.

5.2 Dead code

The optimizer has the ability to identify and remove dead code. This follows directly from the analyzer's construction of the CFG.

If there is dead code, the CFG will not be a connected graph. Solely the one connected component that contains the entrypoint is live. In C*, all other connected components may be removed because there are no language features that allow jumps across connected components.

Our analyzer starts traversing the code from the entrypoint `0x0`. From there on, all jumps are taken as they are found. Therefore, code that could not be reached by the analyzer must be dead. This condition does not catch all unreachable code (for example, branch conditions are not checked for satisfiability), but a decent subset.

To implement dead code elimination, we simply check instructions for an associated CFG node during our optimization passes. The analyzer gives such a node to every instruction that it visited. Instructions that lack a CFG node may therefore be removed as dead code.

5.3 Dead registers

A similar notion of deadness may be introduced for registers. We say a register is *dead* if its value is not going to be accessed in the future.

Instructions that write to a dead register are a special case of `enop`s. However, our mechanism for finding `enop`s is unable to identify such instructions, so they must be identified in a separate pass.

The analyzer keeps track of the liveness of all registers throughout the code. We can query the analyzer for whether register r is dead at instruction i. This lets us identify instructions that write to dead registers simly by checking whether `rd` is dead for each instruction. If so, we may safely remove the instruction in question.

A minor pitfall exists here in that `ecall` instructions may very well return a value to a register even though an `ecall` does not directly have a target register. This is handled in a special case.

5.4 Peephole optimizations

In the following sections, we will present the peephole optimizations employed by the optimizer. These transformations are directly tailored towards code emitted by selfie.

5.4.1 Pointer arithmetic

To keep the compiler simple, selfie's implementation of pointer arithmetic is forced into juggling constants in several registers. This provides us with an opportunity to apply some form of constant folding.

Typical C* code uses pointer arithmetic to emulate struct-like functionality using constant offsets.

```
uint64_t get_class(uint64_t* entry) {
  return *(entry + 3);
}
```

This code is translated to the following machine code by selfie:

```
0x908(~512): 0x00300313: addi t1,zero,3
0x90C(~512): 0x00800393: addi t2,zero,8
0x910(~512): 0x02730333: mul t1,t1,t2
0x914(~512): 0x006282B3: add t0,t0,t1
```

We can have our pattern matching engine scan binaries for such `addi, addi, mul, add` sequences of instructions. If it turns out that the target registers of the initial `addi`s are dead after the final `add`, we can replace the sequence with a single `addi`.

```
0x908(~512): 0x006282B3: addi t0,t0,24
```

Therefore, we save three instructions by offloading the computation to compile time.

Of course, the pattern matcher must keep track of the values loaded into `t1` and `t2` so we can precompute the result of the optimization. These values are stored in global variables as the pattern is matched. Later, they are referenced to obtain the result of the computation.

While this optimization in its current form can't be applied to instances where the pointer offset is dynamic, the lack of structs in C* means that pointer arithmetic is a commonly used pattern to obtain struct-like functionality. Therefore, even with this limitation, the optimization is still quite commonly applicable.

5.4.2 Return constant

When encountering a procedure that returns a value, selfie will always load it into a temporary register before loading it into the return register `a0`. If the return value is constant, we can skip this intermediate step and load the value directly into the `a0` register.

```
uint64_t is_character_new_line() {
  if (character == CHAR_LF)
    return 1;
  else if (character == CHAR_CR)
    return 1;
  else
    return 0;
}
```

```
0x6874(~2615): 0x00100293: addi t0,zero,1
0x6878(~2615): 0x00028513: addi a0,t0,0
```

```
0x6878(~2615): 0x00028513: addi a0,zero,1
```

5.4.3 Return optimization

For another transformation in the context of `return`s, we can optimize away short sequences of rather useless instructions that selfie sometimes seems to produce near `return`s.

```
uint64_t* allocate_symbol_table_entry() {
  return smalloc(2 * SIZEOFUINT64STAR + 6 * SIZEOFUINT64);
}
```

The corresponding machine code loads the value of `a0` into `t0`, then erases the value in `a0`, and finally loads the value of `t0` back into `a0`, achieving nothing at all.

```
0x7F4(~506): 0x00050293: addi t0,a0,0
0x7F8(~506): 0x00000513: addi a0,zero,0
0x7FC(~506): 0x00028513: addi a0,t0,0
```

Given that `t0` is dead, it can safely be removed completely.

5.4.4 Jumps with offset 1

Selfie will emit a `jal` to the procedure epilogue for *every* `return` statement it sees, even if it's the last statement before the end of the procedure. This leads to an occasional `jal` instruction that merely jumps to the next instruction and is therefore equivalent to a no-op.

```
uint64_t is_not_rbrace_or_eof() {
  if (symbol == SYM_RBRACE)
    return 0;
  else if (symbol == SYM_EOF)
    return 0;
  else
    return 1;
}
```

The optimizer looks for `jal` instructions with a jump offset of 1, as they are produced by the final `return` in the above procedure, and simply removes them.

```
0x85AC(~3228): 0x0040006F: jal zero,1[0x85B0]
```

6 Results

To measure the impact of our optimizations, we added several statistics to selfie's profiler. The profiler keeps track of the percentage of runtime `enop`s as well as the number of reads and writes to registers and memory.

The full profiler output of both an optimized and an unoptimized self compilation of selfie (`./selfie -c selfie.c -m 64 -c selfie.c`) is given in the Appendix. Notable changes include:

- The total number of executed instructions decreased by about 5.6% from $2.69 \cdot 10^8$ to $2.53 \cdot 10^8$ instructions.
- The percentage of `enop`s at runtime decreased by 17.5%, and the total number of instructions executed decreased by 5.6%.
- The savings stem from decreasing the number of init (`lui, addi`), memory (`ld, sd, jal`) and control (`beq, jal, jalr`) instructions. In comparison, the relative number of compute instructions (`add, sub, mul, divu, remu, sltu`) performed increased by 5.1%, meaning the percentage of work compared to I/O increased.
- Even without any transformations geared towards spatial performance, we obtain decreased memory usage by roughly half a percent (1.85MB to 1.84MB).

Meanwhile, there were very little changes to the ratio of reads to writes for memory and registers.

6.1 Performance of individual optimizations

These measurements were taken from selfie's profiling output when running a self compilation. The optimizations were applied beforehand to the `selfie` binary. Each row shows the performance metrics for each transformation applied in isolation.

Optimization	Executed instructions	Percentage of nops	Sites
Baseline	269136740	20.41	n/a
pointer arithmetic	268899164	20.36	92
return constant	268476405	20.18	148
effective nop	267499515	19.89	2463
return optimization	263395622	19.71	45
nop jal	262618548	18.40	195
total	253974453	16.82	3125

6.2 Conclusion

In total, about 20% of the potential for optimization is realized as far as instructions that have no effect on the machine state go. This is of course a small subset of all optimization potential, however it is one that is accurately measurable.

In terms of time savings, 16 million (~5.63%) less executed instructions translates to 1.2 seconds (~5.00%) faster self compilation on mipster running on an Intel Core i7-8565U.

```
$ time ./selfie -l selfie_optimized -m 64 -c selfie.c
[...]
22.98s user 0.08s system 99% cpu 23.076 total

$ time ./selfie -l selfie_vanilla -m 64 -c selfie.c
[...]
24.17s user 0.11s system 99% cpu 24.291 total
```

The optimizations that turned out to have the largest impact on selfie's self compilation are those related to procedure calls, which makes intuitive sense given that selfie is a single-pass compiler with a recursive descent parser. An especially large number of procedure calls can be expected in this case, similarly, there's little traffic on the heap and more on the stack; therefore small improvments pertaining to calls will have an outsized impact and the inverse holds for improvements to heap accesses.

For example, the `nop jal` optimization removes a single instruction from procedures with a return value. Applying this optimization in not even 200 places reduced the number of executed instructions by roughly 6.5 million.

On the other hand, the pointer arithmetic optimization removes three instructions in each of its ~100 sites, however the impact ends up being a mere ~200,000 less executed instructions.

It is important to observe that due to the aforementioned characteristics of selfie's compiler, our benchmark workload may not be representative of other workloads, and so it may turn out that other optimizations are more effective for workloads with different peculiarities.

7 Appendix

7.1 Unoptimized profiler output

Profiler output after running an unoptimized self compilation.

```
./selfie: summary: 269136740 executed instructions [20.41% nops]
        and 1.85MB(2.90%) mapped memory
./selfie: init:    lui: 789142(0.29%), addi: 113934591(42.33%)
./selfie: memory:  ld: 58018000(21.55%), sd: 37568230(13.95%)
./selfie: compute: add: 6813372(2.53%), sub: 5422493(2.01%),
        mul: 6562695(2.43%)
./selfie: compute: divu: 2007533(0.74%), remu: 2006589(0.74%)
./selfie: compare: sltu: 4064802(1.51%)
./selfie: control: beq: 5531922(2.05%), jal: 17536534(6.51%),
        jalr: 8601460(3.19%)
./selfie: system:  ecall: 279377(0.10%)
./selfie: profile: total,max(ratio%)@addr,2max,3max
./selfie: calls:   8601460,2357824(27.41%)@0x2D18,1186412(13.79%)
        @0x2F2C,1105319(12.85%)@0x2E64
./selfie: loops:   455934,153051(33.56%)@0x3B94,122749(26.92%)
        @0x719C,105501(23.13%)@0x6988
./selfie: loads:   58018000,2357824(4.06%)@0x2D2C,2357824(4.06%)
        @0x2D30,2357824(4.06%)@0x2D40
./selfie: stores:  37568230,2357824(6.27%)@0x2D1C,2357824(6.27%)
        @0x2D24,1186412(3.15%)@0x2F30
./selfie: CPU+memory:    reads+writes,reads,writes[reads/writes]
./selfie: heap segment:  4283172,4036588,246584[16.37]
./selfie: gp register:   12750664,12750663,1[12750663.00]
./selfie: data segment:  12907920,11968375,939545[12.73]
./selfie: ra register:   33847092,16923546,16923546[1.00]
./selfie: sp register:   184545748,119650915,64894833[1.84]
./selfie: s0 register:   56927541,40283368,16644173[2.42]
./selfie: stack total:   275320381,176857829,98462552[1.79]
./selfie: stack segment: 78395571,42013382,36382189[1.15]
./selfie: a0 register:   24711094,7987635,16723459[0.47]
./selfie: a1 register:   260187,0,260187[0.00]
./selfie: a2 register:   260187,0,260187[0.00]
./selfie: a3 register:   1,0,1[0.00]
./selfie: a6 register:   1,0,1[0.00]
./selfie: a7 register:   279377,0,279377[0.00]
./selfie: args total:    25510847,7987635,17523212[0.45]
./selfie: t0 register:   118783627,59401408,59382219[1.00]
./selfie: t1 register:   53923458,26980916,26942542[1.00]
./selfie: t2 register:   11291618,5645809,5645809[1.00]
./selfie: t3 register:   373810,186905,186905[1.00]
./selfie: t4 register:   77440,38720,38720[1.00]
./selfie: t5 register:   61952,30976,30976[1.00]
./selfie: t6 register:   15488,7744,7744[1.00]
./selfie: temps total:   184527393,92292478,92234915[1.00]
```

7.2 Optimized profiler output

Profiler output after running an optimized self compilation.

```
./selfie: summary: 253974453 executed instructions [16.82% nops]
        and 1.84MB(2.88%) mapped memory
./selfie: init:    lui: 789188(0.31%), addi: 105553856(41.56%)
./selfie: memory:  ld: 58039393(22.85%), sd: 37582056(14.79%)
./selfie: compute: add: 6704220(2.63%), sub: 5424433(2.13%),
        mul: 6453836(2.54%)
./selfie: compute: divu: 2008460(0.79%), remu: 2007217(0.79%)
./selfie: compare: sltu: 4065917(1.60%)
./selfie: control: beq: 5533351(2.17%), jal: 10928393(4.30%),
        jalr: 8604603(3.38%)
./selfie: system:  ecall: 279530(0.11%)
./selfie: profile: total,max(ratio%)@addr,2max,3max
./selfie: calls:   8604603,2358768(27.41%)@0x25F0,1186881(13.79%)
        @0x276C,1105793(12.85%)@0x26AC
./selfie: loops:   456127,153054(33.55%)@0x32AC,122749(26.91%)
        @0x5F60,105501(23.12%)@0x5868
./selfie: loads:   58039393,2358768(4.06%)@0x2604,2358768(4.06%)
        @0x2608,2358768(4.06%)@0x2618
./selfie: stores:  37582056,2358768(6.27%)@0x25F4,2358768(6.27%)
        @0x25FC,1186881(3.15%)@0x2770
./selfie: CPU+memory:    reads+writes,reads,writes[reads/writes]
./selfie: heap segment:  4284432,4037672,246760[16.36]
./selfie: gp register:   12754416,12754415,1[12754415.00]
./selfie: data segment:  12911879,11972327,939552[12.74]
./selfie: ra register:   33859358,16929679,16929679[1.00]
./selfie: sp register:   184613999,119695244,64918755[1.84]
./selfie: s0 register:   56948932,40298779,16650153[2.42]
./selfie: stack total:   275422289,176923702,98498587[1.79]
./selfie: stack segment: 78425570,42029745,36395825[1.15]
./selfie: a0 register:   17995635,6044714,11950921[0.50]
./selfie: a1 register:   260340,0,260340[0.00]
./selfie: a2 register:   260340,0,260340[0.00]
./selfie: a3 register:   2,0,2[0.00]
./selfie: a6 register:   1,0,1[0.00]
./selfie: a7 register:   279529,0,279529[0.00]
./selfie: args total:    18795847,6044714,12751133[0.47]
./selfie: t0 register:   113379282,56719511,56659771[1.00]
./selfie: t1 register:   52708202,26767325,25940877[1.03]
./selfie: t2 register:   11072829,5536450,5536379[1.00]
./selfie: t3 register:   373876,186938,186938[1.00]
./selfie: t4 register:   77440,38720,38720[1.00]
./selfie: t5 register:   61952,30976,30976[1.00]
./selfie: t6 register:   15488,7744,7744[1.00]
./selfie: temps total:   177689069,89287664,88401405[1.01]
```

References

[1] F. E. Allen and J. Cocke. 1971. *A catalogue of optimizing transformations.* IBM Thomas J. Watson Research Center. Retrieved from https://books.google.at/books?id=oeXaZwEACAAJ

[2] C. Kirsch. C* language specification. Retrieved from https://github.com/cksystemsteaching/selfie/blob/main/grammar.md

[3] C. Kirsch and others. Selfie git repository. Retrieved from https://github.com/cksystemsteaching/selfie

[4] John Markoff. 2005. Writing the fastest code, by hand, for fun: A human computer keeps speeding up chips. *New York Times* (November 2005). Retrieved from https://www.nytimes.com/2005/11/28/technology/writing-the-fastest-code-by-hand-for-fun-a-human-computer-keeps.html

[5] Henry Massalin. 1987. Superoptimizer - A look at the smallest program. In *Proceedings of the second international conference on architectural support for programming languages and operating systems (ASPLOS II)*, ACM Press, 122–126. Retrieved from https://dl.acm.org/citation.cfm?id=36194

[6] John Regehr. 2012. ARM math quirks on raspberry pi. Retrieved from https://blog.regehr.org/archives/793

[7] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. *CoRR* abs/1711.04422, (2017). Retrieved from http://arxiv.org/abs/1711.04422

[8] H. S. Warren. 2003. *Hacker's delight.* Addison-Wesley. Retrieved from https://books.google.at/books?id=iBNKMspIlqEC

[9] Thomas Wulz. 2021. Linear-time static analysis of RISC-v binary code. Bachelor's Thesis.