Evaluating Souper: A Synthesizing Superoptimizer

Emirhan B. Demir
Supervisors: Dennis Sprokholt, Soham Chakraborty
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Modern compilers exploit syntax & semantics to optimize input programs. Often such optimization rules are arduous to get right and the output is not guaranteed to be globally optimal. Superoptimizers take a different approach to this problem by traversing the program space. This study focuses on Souper, a synthesizing superoptimizer which makes use of an enhanced counter-example-guided inductive synthesis loop to find optimizations. We first detail the working mechanism of the superoptimizer and its components, then we explain our attempts at reproducing the results mentioned by Souper's authors. Finally, we give three program classes each exercising different aspects of the superoptimizer and how these are useful in gaining insight into Souper's optimization capabilities and use cases.

1 Introduction

Most modern compilers have an optimization step where the compiler attempts to optimize the input program based on syntax and semantics. These optimization rules are written manually and more often than not require great amounts of intellectual effort. A superoptimizer tries to enhance this optimization step with a different approach. Instead of achieving optimizations based on the input program's syntax and semantics it traverses the program space to find a valid and globally optimized solution.

The idea of superoptimization is not a new concept, first attempts to enhance or replace optimization passes in conventional compilers date back to 1987 [10]. There have been a number of approaches to find optimized programs that compilers cannot achieve with optimizations strategies they currently make use of [4], [6].

Still, since superoptimizers are not yet in widespread use we lack insight into their optimization capabilities and suitable use cases. In this study we are focusing on Souper [13], a synthesizing superoptimizer.

At the first glance Souper's superoptimization process may seem like it comprises many free-flowing components. The relationships between these components and the superoptimization process can be summarized as follows:

- Compiling from source to LLVM [8] IR.
- Extracting candidate optimizations from LLVM IR.
- Synthesizing optimizations using CEGIS [6].
- Incorporating these in LLVM.

Given a program, using an LLVM front-end we can extract LLVM IR which Souper can use to find optimization candidates. Once these candidates are found Souper utilizes an improved CEGIS loop to synthesize a potentially optimized and equivalent program. Finally, the optimization can be incorporated back into LLVM to be outputted as binary. The steps given above are explained in more detail in the sections to follow.

This study attempts to reproduce results presented by Souper's authors and aims to find the program classes where Souper can infer optimizations that a classical optimizing compiler might fail to achieve. The research questions answered can be phrased as:

- Can the original results obtained with Souper be reproduced?
- What program classes does Souper work best in?

To answer these questions, it is first necessary to reproduce the results mentioned in the original paper to have a baseline of Souper's optimization capabilities. Then, we devised three program classes that test Souper's specific features and components. These three program classes are programs with high cyclomatic complexity, programs that Souper cannot fully extract from, and finally those with undefined behaviour. For all the programs listed above the metric for evaluating the optimization result is the size of the output binary and the resulting Souper intermediate representation.

2 Intermediate Representations

Both LLVM's and Souper's intermediate representations are referenced quite frequently throughout this paper. Therefore, we believe that detailing both intermediate representations -especially Souper's- further would benefit establishing some common ground and defining the necessary primitives the rest of the paper makes use of.

2.1 LLVM

Low Level Virtual Machine compiler infrastructure [8] can be divided into 3 main components. A front-end for a certain programming language which takes source code as input and outputs LLVM intermediate representation. Second, an optimizing mid-end which uses the extracted intermediate representation, applies `opt` passes and outputs optimized IR. Finally, an instruction set specific back-end which converts the optimized IR into architecture specific instructions that can be executed. This architecture allows LLVM to support a plethora of programming languages and CPU architectures while still having a single optimizing component. A notable feature of LLVM's intermediate representation is control flow through ϕ nodes. Souper also makes use of these nodes in order to reason about the incoming control flow within a program.

```c
int foo(bool cond, int z) {
    int x, y;
    if (cond) {
        x = 3 * z;
        y = z;
    } else {
        x = 2 * z;
        y = 2 * z;
    }
    return x + y;
}
```

```llvm
define i32 @f(i1 %0, i32 %1) {
br i1 %0, label %3, label %5
label %3:
%4 = mul nsw i32 %1, 3
br label %8
label %5:
%6 = shl nsw i32 %1, 1
%7 = shl nsw i32 %1, 1
br label %8
label %8:
%.07 = phi i32 [ %4, %3 ], [ %6, %5 ]
%.0 = phi i32 [ %1, %3 ], [ %7, %5 ]
%9 = add nsw i32 %.07, %.0
ret i32 %9
}
```

Table 1: Listings showing a simple function and the corresponding LLVM IR [13].

To illustrate LLVM's intermediate representation better, Table 1 shows a simple function written in `C` and the corresponding LLVM IR. As mentioned before, it is worth noticing how conventional control flow is translated into ϕ nodes with corresponding predecessors.

2.2 Souper

Souper's intermediate representation is extracted from LLVM IR and exhibits similar features. Besides having an understanding of the intermediate representation itself in addition to how it is extracted gives valuable insight into inner workings, capabilities, and limitations of the superoptimizer.
Souper's intermediate representation closely follows that of LLVM's. In total it has 51 instructions that are derivations from LLVM's integer and scalar instruction subset [13]. Additionally, Souper provides 10 LLVM intrinsics as instructions. These include overflow checks for arithmetic, hamming weight calculations, and a byte swap instruction.

For each integer-typed value returning LLVM module Souper's extractor creates a root node for a candidate optimization. Each candidate optimization is made of a left and a right hand side where the left-hand side contains the extracted IR and the right-hand side contains the optimized version. In order to complete the rest of the left-hand side of a root node it recursively follows the data flow path back [13]. During its backwards traversal the extractor adds path conditions and `blockpc` constructs as it encounters ϕ nodes and branches. Once the extractor encounters a function return value or a function entry point extraction terminates. Similarly, extraction stops if the extractor arrives at an instruction that Souper does not have a representation for, such as a load from memory or a floating point operation.

```
%0 = block 2
%1:i32 = var
%2:i32 = shlnsw %1, 1:i32
%3:i32 = phi %0, %1, %2
%4:i32 = mulnsw 3:i32, %1
%5:i32 = phi %0, %4, %2
%6:i32 = addnsw %3, %5
infer %6
→
%7:i32 = shl %1, 2:i32
result %7
```

Table 2: Listing showing Souper IR for the example in Table 1. The part until the right-arrow representing the left hand side and the part after showing the optimization found [13].

Table 2 illustrates the extracted Souper IR from the function shown in Table 1. It is worth mentioning the similarity of the instructions present and how LLVM intrinsics such as `nsw` (no signed overflow) are incorporated into instructions themselves.

3 Program Synthesis

Program synthesis is the automated construction of software and has been one of the holy grails of software engineering [7]. Synthesis relieves the programmer from the burden of describing how the problem should be solved instead it only requires the programmer to give a specification of the program. The synthesizer then, generates a program that provably satisfies this specification [4]. More formally, a synthesizer is a solver for second order existential logic. Consider the formula

$$\exists \mathbf{P} \cdot Q\mathbf{x} \cdot \sigma(\mathbf{P}, \mathbf{x})$$

where **P** ranges over functions, Q is a quantifier, **x** ranges over ground terms and σ is a quantifier-free formula. A synthesizer generates a model that can map each second-order variable P to some function of appropriate type [4].
Over the years program synthesis had found many practical use cases, including but not limited to: generating optimal code sequences, automating repetitive programming tasks, and filling in low-level details for some higher level specification. In order to explain the synthesis method Souper follows, it is necessary to establish some preliminaries. To this end, before detailing Souper's synthesis process this section explains:

- Deductive vs. Inductive Synthesis
- Counter-example-guided Abstraction Refinement
- Counter-example-guided Inductive Synthesis

Deductive vs. Inductive Synthesis

If a program specification is complete, in the sense that it does not require iterative revision and tuning it is said to be a deductive synthesis process [4]. Unfortunately, such specifications are usually not available or are hard to write, on the other hand what is in reach most of the time are incomplete program descriptions, desirable and undesirable behaviour, input/output examples and so on. In contrast to deductive synthesis, inductive synthesis methods can makes use of common patterns in given facts such as these. Even though this offers great flexibility, the resulting program may not be sound in the sense that it may exhibit incorrect behaviour in cases the specifications failed to cover completely.

Counter Example Guided Abstraction Refinement

In order to restore soundness of resulting programs a counter example guided synthesis approach follows a similar pattern to counter example guided abstraction refinement (CEGAR) [3]. The main working mechanism of CEGAR can be regarded as three steps. First, for a given program an abstraction corresponding to this program is extracted, then it is checked whether this is the abstraction formula to be reached. If this check reveals a counterexample this counterexample is then assumed to be also in the unabstracted structure. If this is the case an actual counterexample is found and the process returns otherwise the counterexample is determined to be spurious and the abstraction is refined in the last step. In the last step of the process the abstraction is modified so that it does not accept the spurious counterexample found in the previous step anymore. After refinement process jumps back to the second step of finding counterexamples.

Counter Example Guided Inductive Synthesis

Similarly, counter example guided inductive synthesis (CEGIS) [4], [6] runs in a loop where each iteration inductive generalization is attempted based on the counter examples provided by a verification oracle. First, the program space is searched for a candidate P that satisfy

$$\exists \mathbf{P} \cdot \forall \mathbf{x} \in INPUTS \cdot \sigma(\mathbf{P}, \mathbf{x})$$

then this candidate program is passed to the verification oracle which tries to find input distinguishing P from the global solution that maps all inputs correctly. If such an input is found it is added to the set of inputs, if not the process terminates and the program is

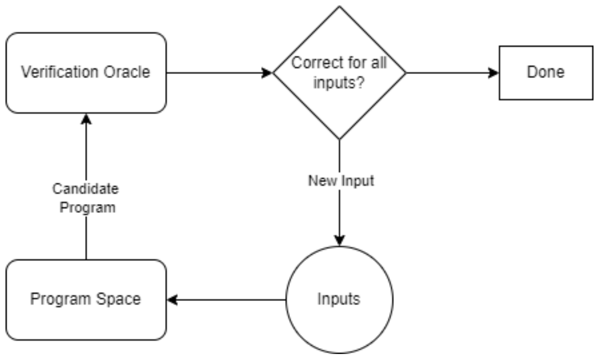

Figure 1: Simplified counter example guided inductive synthesis loop.

returned. This way the process essentially synthesizes programs that works for more and more inputs as illustrated in Figure 1.
In general, an SMT solver is used as the verification oracle in such a synthesis process. As defined in [5], "Satisfiability modulo theories (SMT) generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in these theories.".

Besides the synthesis itself the program space searched is also crucial in the overall efficiency of the synthesizer. Even though for a finite program space synthesis is guaranteed to terminate by either returning the correct program or exhausting the search space, it might take a considerable amount of time. Therefore, two important aspects to consider for a system that makes use of this synthesis method are potential ways to shrink the program space and the manner in which this space is traversed.

3.1 Souper's Synthesis Process

Program synthesis lies at the heart of Souper's superoptimization process. Once an optimization candidate is extracted from LLVM IR as a left hand side, the superoptimizer synthesizes a potentially optimized right hand side. In order to do so, an enhanced version of the CEGIS loop is used [13]. Since synthesis usually involves large search spaces, to keep superoptimization scalable and efficient the portion of the input that is going to be searched and the number of times the verification oracle is queried should be reduced as much as possible. Even though CEGIS does not attempt to naively enumerate the entire program space and performs well in most cases, Souper makes use of various strategies to shrink the search space at hand [12].

3.1.1 An outer CEGIS loop

One relatively straightforward strategy Souper uses in order to enhance the runtime and effectiveness of CEGIS is to wrap the synthesis in an outer loop [13]. This way Souper can put constraints on the output RHS. Doing so, Souper first attempts to synthesize right hand sides with no new instructions and which iteration this constraint is relaxed by one.

3.1.2 Pruning without Dataflow Analysis

A more substantial step up in synthesis efficiency is possible through pruning the search space. Souper attempts to do so in a multitude of methods. Ones mentioned here do not make use of dataflow facts Souper can extract and instead rely on more general rules and properties [12].

- **Synthesizing constants directly** As opposed to trying to verify potential optimizations with constants directly, Souper leaves them as symbols and once a correct optimization is found the symbols are synthesized to constant directly.
- **Using a cost model** Souper utilizes a cost model to prune potential right hand sides that are more expensive than the left hand side specification.
- **Ad hoc pruning strategies** Souper applies various pruning strategies such as exploiting commutativity, not generating instructions with all constant operands etc.

3.1.3 Pruning with Dataflow Analysis

Given the specification

$$f(x) = (x * x * x)|1$$

consider the partially symbolic candidate

$$g(x) = H(x) << C$$

Where H represents a directed acyclic graph of arbitrary and not enumerated instructions or a so-called *hole* the result of which is bit shifted to the left by a constant C [12]. If such a candidate can be eliminated, the amount of times the verification oracle needs to be queried goes down considerably.

Based on the specification a dataflow analysis can reason that the least significant bit of the result is always set because of the *bitwise-or* operation. Similarly, looking at the candidate it can be concluded that, since a bit shift by zero is meaningless the least significant bit of g is always cleared. These dataflow facts clearly conflict. Therefore this branch of the search tree can be pruned [12].

In other words, the pruning problem is: given a specification f and a partially symbolic candidate g is it possible to prove that there is no concrete instantiation of g that refines f [12]? To answer this question Souper searches for a fact about f that conflicts with at least one fact that is true for all instantiations of g. Souper looks for two kinds of conflicts:

- **Root conflicts**
  Souper uses three forward dataflow analysis to look for root conflicts [12]:
  - **Known bits**
    Attempts to prove that each output bit is always either 0 or 1
  - **Integer ranges**
    Attempts to prove that the output is within a range of integer values
  - **Bivalent bits**
    Attempts to prove that each output bit can be flipped by choosing two different input values

- **Leaf conflicts**
  Souper uses three backward dataflow analysis to look for root conflicts [12]:
  - **Required bits**
    Attempts to prove that individual input bits influence the output
  - **Don't care bits**
    Attempts to prove that individual input bits do not influence the output
  - **Forced bits**
    Tracks individual bits in symbolic constants which are forced to be a particular value, given a specific output value

4 Methodology

As introduced previously, this paper aims to evaluate Souper by reproducing its original results and devise different program classes to test its optimization performance. In order to have meaningful results and draw consistent conclusions, we have decided upon common evaluation metrics. These metrics are threefold: output Souper IR, binary size, and compilation speed.

In order to see how well Souper can actually extract its IR and find possible optimizations, it is crucial to examine resulting IR of optimization candidates found. In addition to showing how well Souper can handle specific program cases this method also casts some light on the working details of the superoptimizer and its capabilities. This metric also allows Souper to be evaluated independently, in contrast to merely using it as a compiler extension.

When it is incorporated with the compilation process, however how successful the superoptimizer may be to be feasible as a compiler aid it must not increase output binary size drastically.

At the same time, when Souper is used as an online optimizer due to the fact that it adds an additional pass to LLVM's optimization process, compilation is expected to take longer, still, compilation speed is a useful metric to provide insight into feasibility of incorporating Souper into compilation. The last two of the above-mentioned metrics are also employed in the original paper [13], this makes the comparison process with the original results more straightforward.

5 Reproducing Original Results

In order to have a preliminary understanding of Souper's inner workings and capabilities we attempted to reproduce the results presented in the original paper[13]. In other words, the aim is to see whether under the same experimental conditions, the same results can be obtained consistently.

It is important to see Souper's optimization synthesis with some of the small test programs as well as to see how Souper's capabilities may be useful for a project of a larger scale. To this end, the optimization results presented in the paper for small programs and for `clang-3.9.0` are reproduced.

5.1 Attempts at Compiling `clang`

In order to test Souper's optimizing capabilities on a large program the authors of the original paper had decided to compile LLVM with its `C` frontend `clang` [13]. The `clang` codebase is around 2,5 million lines of code and according to their findings a clean build without Souper takes around 13 minutes. Additionally, the authors have replicated the average workflow of an LLVM developer by running an incremental build of `clang` daily. Since on most days there are changes to widely included headers a complete build is necessary and the build time usually stays the same. On the other hand when they included Souper in the compilation process a clean build took 88 minutes. This is considerably longer than normal build times, to mitigate this Souper caches potential optimizations and the corresponding right-hand sides which reduces the compilation times of incremental builds down to around 9 minutes.

Even though the above mentioned results seem straightforward, unfortunately they are not as easily reproducible as one might expect. In order to have a better insight into the attempt of reproducing the original results it is worth discussing different types of utility Souper provides.

5.1.1 Standalone Souper

The obvious way of using the superoptimizer is as its own executable. Given an LLVM bitcode file, independent of any other program or process Souper can output the optimizations it finds in its intermediate representation. Even though this might seem satisfactory, for compiling larger programs with a large number of object files, this method quickly becomes unwieldy.

5.1.2 Dynamically linking to LLVM's `opt` pass

After being generated by a front-end, LLVM IR is optimized in LLVM's mid-end. Souper as a shared library can dynamically link to the optimization pass LLVM offers. This way optimizations Souper may find are automatically applied to the input.

5.1.3 Drop-in Compiler

Finally, Souper offers a drop-in compilers that effectively replace `clang` and `clang++`. This way, a source file can be directly compiled into a binary with the optimizations Souper had found applied. Additionally, Souper can cache the potential optimizations without inferring a right-hand side, this can be followed by inferring optimizations on the cache, overall this offers a clear division in the compilation process and makes it more manageable. The drop-in compilers make the process of incorporating superoptimization into the build process of larger programs easier.

The obvious way moving forward may seem like using the drop-in compiler to build the latest version of `clang` with LLVM and a back-end but after our attempts with different configurations such as the inclusion of an external cache, the compilation process was never successful. An obvious culprit in this case is, since the original results there have been substantial development on `clang`, LLVM, and Souper itself. This creates a problem with versions of the programs used. While the original was being written Souper would be linked against `clang` 3.9 to be used as a drop-in compiler and the original experiments were conducted on LLVM version 3.9's codebase. In order to replicate this, we have tried to link

Souper against LLVM 3.9 but unfortunately this was not possible. Still, after linking Souper with the newer toolchain we have tried to build a `clang` 3.9 binary again using the drop-in compiler. When an external cache was used with this setup a clean build was possible, which took around 51 minutes, yet this was most likely a one time fluke since it was not reproducible with an identical setup later on. Unfortunately, except for the single occasion mentioned, Souper got stuck and compilation was never finished.

Next, we have tried to split the compilation process into three steps in order to separate the concerns of extracting the potential optimizations and finding right-hand sides for these optimization candidates. In order to do so, we have built `clang` using the drop-in compiler with the `SOUPER_NO_INFER` flag which stops the superoptimizer from inferring right-hand sides for potential optimizations instead these candidates are only cached. Next, optimizations were inferred on the cached candidates. Unfortunately, in this step we have realized that the number of left-hand sides were around 279 000 which -assuming an average RHS synthesis time of 15 seconds- would take around 45 days to fully infer optimizations from. In the paper the number of extracted left-hand sides are mentioned to be around 17 thousand unfortunately the authors have not detailed the necessary compiler configuration for such a result.

These attempts were also repeated using the Docker images [11] supplied in the Souper repository [1] sadly the outcomes were the same.

5.2 Superoptimizing Program Samples

In addition to compiling `clang` the authors also make use of smaller example programs in order to demonstrate Souper's workings with specific program cases. One program explained in detail is said to exploit correlated ϕ nodes in LLVM-IR through `blockpc` statements and is able to fully optimize a switch statement away, this program and the optimization found are shown in Table 4 and Table 5. We thought replicating these programs as well would be beneficial in gaining insight into Souper's capabilities. Given that program at hand is minute, replicating the results may seem extremely straightforward as merely running Souper on the bitcode representation of the source. Unfortunately, once again, this is proven not to be the case. In our experiments with its default build, Souper is not able to find any potential optimizations for this program. Since there were no details given about the process of getting this result we have looked into other projects that have made use of Souper to see the configuration they have used. One such project is Slumps [2] which targets WebAssembly. Still, the flags its authors have used have proven to be somewhat useful with this example program too. With these flags Souper was able to find one (1) potential optimization which is shown in Table 3. Obviously this optimization is neither useful nor the same as the original result.

```
< %4 = urem i32 %3, 4
---
> %4 = and i32 3, %3
```

Table 3: Diff of two LLVM IR files showing the optimization Souper had found. Top shows the original file, bottom with the changes from Souper.

In conclusion, unfortunately, we were not successful in consistently reproducing the results presented by the authors of Souper.

6 Evaluating Souper on Different Program Classes

Besides reproducing original results, we have also solicited different program classes that test Souper. These program classes are devised to exercise or highlight different mechanisms within Souper. We believe that these trials will give more insight into Souper's capabilities and limitations. Even though due to a lack of time these experiments were not fully conducted we provide enough detail to sufficiently narrow down these program types so that such a program is trivially constructable.

6.1 Programs with High Cyclomatic Complexity

One of the prominent features that make Souper's intermediate representation different from LLVM IR is how branching and control flow are represented. Programs that introduce a large amount of branching are ideal in trialing this facet of Souper. In order to enumerate this trait we have made use of the static analysis metric cyclomatic complexity.

6.1.1 Cyclomatic complexity

Cyclomatic complexity is a measure of number of basic paths that can be taken through a program. These basic paths when combined will generate every possible path that can be taken. Even though this metric was developed as a way to measure the testability, maintainability, and code quality of a given piece of software, it is still useful for generalizing to a program class that can be used to test the superoptimizer.

6.1.2 Exploiting correlated ϕ nodes

Each ϕ node in each block in LLVM IR returns the value that corresponds to this block's predecessor. Even though Souper doesn't replicate this behaviour directly, it makes use of a block type value to store information about correlated ϕ nodes.

Normally, once multiple control flow paths converge information about the path conditions for these are lost even if they are useful. Using the block type introduced above and `blockpc` instructions Souper can retain this information and reason about incoming control flow.

```
unsigned foo(unsigned a) {
    switch (a % 4) {
        case 0:
            a += 3;
            break;
        case 1:
            a += 2;
            break;
        case 2:
            a += 1;
            break;
    }
    return a & 3;
}
```

Table 4: Listing showing a function where information is lost after control flow paths merge [13].

Take the listing in Table 4 for example, here information about the remainder of "a" divided by 4 is useful once control encounters the return statement but at this point this information is already lost since the control flow paths have merged. Using `blockpc` constructs Souper can make use of this information as shown in Table 5. As illustrated Souper extracts 6 `blockpc` instructions and can optimize the return value of the above function to a constant.

```
%0 = block 4
%1:i32 = var
%2:i32 = urem %1, 4:i32
%3:i1 = ne 0:i32, %2
%4:i1 = ne 1:i32, %2
%5:i1 = ne 2:i32, %2
blockpc %0 0 %3 1:i1
blockpc %0 0 %4 1:i1
blockpc %0 0 %5 1:i1
blockpc %0 1 %2 2:i32
blockpc %0 2 %2 1:i32
blockpc %0 3 %2 0:i32
%6:i32 = add 1:i32, %1
%7:i32 = add 2:i32, %1
%8:i32 = add 3:i32, %1
%9:i32 = phi %0, %1, %6, %7, %8
%10:i32 = and 3:i32, %9
infer %10
→
result 3:i32
```

Table 5: Listing showing the extracted Souper IR and the optimization found from the listing in Table 4 [13]. The part until the right-arrow representing the left hand side and the part after showing the optimization found

Even though the example above might seem contrived, it is useful in demonstrating Souper's behaviour with programs that use facts about control flow later in the execution. Such occurrences are expected to be more frequent once more branching is introduced which -with high probability- will also drive cyclomatic complexity high.

6.2 Programs Souper cannot fully extract from

As described in section 2 Souper's extractor terminates once it encounters an instruction Souper does not have a representation for. We believe that programs that are heavy in these instructions would be an interesting case to test Souper on. Even though this might seem counter-intuitive at first due to the fact that it does not necessarily play into the strengths of the superoptimizer, such programs constitute a considerable part of most code bases. Therefore, we believe that even though Souper might not necessarily perform the best it can with these programs, these provide valuable insight into how Souper would perform in "real-life" scenarios.

6.3 Programs with Undefined Behaviour

Much like the name suggests programs with undefined behaviour[1] either contain or execute pieces of code that the language standard does not prescribe. Almost always this is not a desired situation to be in. Unfortunately, even in the most well written codebases cases of undefined behavior are present.

6.3.1 LLVM & Souper's representations of undefined behaviour

LLVM represents undefined behaviour in three different ways.

- **Immediate undefined behaviour** caused by actions such as bad memory accesses.
- **undef** value used for uninitialized register/memory locations and can hold any value of its type.
- **poison** value which can turn the return values of `phi` and `select`s to `poison` if one of their inputs is `posion`.

```
int x;
if (cond)
    x = f();
if (cond2)
    g(x);
```

```
entry:
br %cond, %ctrue, %cont
ctrue:
%xf = call @f()
br %cont
cont:
%x = phi [ %xf, %ctrue ], [ undef, %entry ]
br %cond2, %c2true, %exit
c2true:
call @g(%x)
```

Table 6: Listings showing a program that can invoke undefined behaviour if a condition evaluates to false [9].

A program that can cause undefined behaviour and the corresponding LLVM intermediate representation is shown in Table 6. If the first condition evaluates to false the program passes an uninitialized value to a function invoking undefined behaviour. As one might expect this is represented as an `undef` in a ϕ block in LLVM.

Souper takes a slightly different approach to mapping undefined behaviour [13]. Due to the lack of a memory model in Souper IR and since `undef` values mostly surface when dealing with memory, Souper does not have a direct mapping for this kind of undefined behaviour. Immediate undefined behaviour on the other hand is reduced to only one case which is a division by zero. Lastly, `poison` values are represented similarly to LLVM, each `phi` or `select` in Souper IR only propagates undefined behaviour via its selected branch.

It has been found that undefined behaviour can be benignly compiled by LLVM [13]. In these cases Souper's exploitation of undefined behaviour might result in the application not working properly. Therefore, we believe this to be an interesting program class that can introduce a layer of complexity in how superoptimization should happen when undefined behaviour is concerned and in particular how Souper handles theses programs.

[1]https://en.cppreference.com/w/cpp/language/ub

7 Responsible Research

When the subject at hand is program synthesis and optimization the ethical implications may be harder to point out. This study is no exception. The experiments conducted are deterministic except for hardware dependent statistics such as compilation speed. We also provide satisfactory detail to make these experiments fully reproducible.

Apart from input programs, the superoptimizer does not rely on any user input. One issue that is worth addressing concerning the input from the user is the *correctness* of the output program. Given an input program the superoptimizer is not expected to change the behaviour in any shape or form. In other words the input and the output programs should be equivalent in functionality. This is covered by the superoptimizer's synthesis procedure. Since each optimization candidate is compared in terms of behavior to the specification provided the correctness of the input is preserved.

8 Conclusions and Future Work

Writing optimizing compilers and optimization rules is hard to get right and burdensome. Superoptimizers take a different approach to this problem by searching the program space looking for a globally optimal program. In this paper we have evaluated Souper which makes use of program synthesis to solve the optimization problem. We have tried to find out whether original results presented by Souper's authors were reproducible and devised different program classes to determine what kinds of programs Souper works the best with.

To provide better insight into Souper's working principle we have described the intermediate representations it makes use of, we provided an introduction to program synthesis and how Souper synthesizes programs in particular. During Souper's synthesis method we have touched upon how the superoptimizer uses certain techniques to reduce the size of the search space traversed in its inductive synthesis loop.

Then, we have presented the utility Souper offers and our attempts at reproducing the original results put forward by the authors of the superoptimizer. Unfortunately, we were not successful in consistently reproducing these results. All the same, we have devised three program classes each exercising a different component or mechanism within Souper. We believe trialing the superoptimizer on these program classes would provide useful insight into its optimization capabilities, limitations, and use cases.
Given what is being presented possible future work to build on this paper is rather clear. To begin with, even though we have not managed to reproduce the original results, we have presented the utility available and approaches we have already attempted. These themselves provide a baseline on future attempts to reproduce results. Additionally, we provide enough detail about the mentioned program classes to construct concrete programs that belong to these. Which could be used to evaluate Souper in a more empirical manner.

9 Acknowledgements

We thank John Regehr and Manasij Mukherjee for their help in setting up Souper.

References

[1] souper. `https://github.com/google/souper`.

[2] Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. *Superoptimization of WebAssembly Bytecode*, page 36â40. Association for Computing Machinery, New York, NY, USA, 2020.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[4] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337â340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *SIGPLAN Not.*, 46(6):62â73, jun 2011.

[7] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 215â224, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[9] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in llvm. *SIGPLAN Not.*, 52(6):633â647, jun 2017.

[10] Henry Massalin. Superoptimizer: A look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5):122â126, oct 1987.

[11] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[12] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. Dataflow-based pruning for speeding up superoptimization. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[13] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. 11 2017.