Compiler Optimizations and HPC

Raghuveer Shivakumar
University of Colorado
raghuveer.s@colorado.edu

Abstract

”Ask not what you can do on an HPC system to make your code run faster - ask what can your HPC system do to help you worry less about having to make your code run faster”. The traditional view of an HPC system has been that of a cluster of unintelligent computers endlessly crunching numbers so that the problem you set out to solve can be solved. But in this report, I argue that HPC is not a unilateral paradigm as it is usually made out to be, but a potentially intelligent and cooperative paradigm that can not only help us use its compute rich resources to solve problems faster, but also help solve these problems more effectively. We try to formulate a new meta-model of HPC, particularly High-Performance Compilers, that can automatically reason about programs without significant human intervention. We first describe the philosophical underpinnings of the problem of self-improvement and then we introduce how compilers can be seen as an epistemic tool that can help us achieve this goal. We then discuss the current state of compilers and compiler optimizations. Finally, we discuss the theoretical and practical implications using the example of a Superoptimizer in an HPC system.

1. Introduction

High-Performance Computing (HPC) is traditionally seen as a paradigm that caters to compute-intensive applications. Unfortunately, this view has led to many misconceptions about the paradigm among the uninitiated, where the invocation of the term HPC leads to imageries of a large grid of interconnected computers churning out incomprehensible numbers in parallel all the time. Although there is some truth to it, my research into HPC for this course says that this is not the entire story. HPC is not this unilateral caricature of a paradigm. HPC is a conglomerate of many different paradigms, large and small, that are being used to solve a number of different problems. When we talk about solving a compute-intensive task, the conversation is usually about creating an efficient algorithm or using a superior compute resource to solve that problem, but this view of HPC only takes into account the problem at hand. In this report, I want to formulate a self-solving model of HPC that can be used to solve the problem at hand and at the same time improve the efficiency with which it solves the given problem. Technically, we will be redefining HPC to incorporate the notion of self-enhancement.

2. Background

2.1 The Philosophical Issue

What leads to self-improvement? An ontologist would say that *understanding* the notion of being leads to self-improvement, whereas an epistemologist would say that *understanding* the nature of knowledge acquisition leads to self-improvement. Whether you agree with the ontologist or the epistemologist, the answer is not immediately obvious from these statements. But there is a common ground between the two, which is that self-improvement in some form or the other involves **”understanding”**. And understanding requires reasoning about concepts.

If we give a nod to the idea that the ability to *reason* captures at least some aspects of what it means to be a self-improving system, we can then extrapolate this

http://dx.doi.org/10.1145/

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet \arrow[r] \arrow[d] & \bullet \arrow[d] \\
\bullet \arrow[u] \arrow[r] & Entity & \bullet \arrow[l] \arrow[d] \\
\bullet \arrow[u] & \bullet \arrow[l] \arrow[u] & \bullet \arrow[l]
\end{tikzcd}$$

Figure 1. Structuralism

to the world of HPC, particularly software, to see what it would mean for a computer to be a self-improving system. At least from a category-theoretic point of view "to reason" would mean that there is some kind of `morphism` between the objects being studied and that the system studying or analyzing it is able to infer these relationships. This is the basis for the idea that a computer can be a self-improving system.

2.2 Reification

To start with, we can enumerate the tools that help us reason about programs — the building block that helps form a cohesive piece of software. And it seems to me that the only tool that can help us reason about programs is a compiler — an epistemic tool that infers the relationship between the object and the meta, or more technically, the data and the code. It gives us the ability to talk about the meaning of a program, or in compiler-speak semantics. This still cannot be called self-improving in strict terms, as the meaning-making ability is still deterministic. But it is still a useful starting point to build a system that is self-improving.

2.3 Compilers and Structuralism

Relationships (or morphisms in general) point toward a broader philosophical concept — The idea of structuralism. In structuralism, everything is strictly defined by its relationship to other things in a given structure. For eg. In Peano arithmetic, the existence of the number 1 is exhaustively defined by its role as a successor to the number 0. Similarly, if we take the ever famous thought experiment of a "Fregean Julius Caesar", where Frege poses this question: "What number would Julius Caesar be if he were to be a number?", we as new converts can answer this question by saying that Julius Caesar can be any number in a given number system(structure) if the relationship is invariant over time. This is the idea of structuralism.

$$\begin{tikzcd}
 & x = 2 & \\
 & valid & \\
x \arrow[uur] & & 2 \arrow[uul] \\
 & is\_valid & \\
 & x == 2 \arrow[uul] \arrow[uur] &
\end{tikzcd}$$

Figure 2. Example of Syntax Semantics Graph

And compilers are naturally structuralistic in nature, that is they enable reasoning naturally by inferring the relationships between the object and the meta.

2.3.1 Syntax-Semantics Graph

To understand this let's think of nodes as syntax and edges as Semantics. And whether a node or an edge can be added, as Syntactic/Semantic analysis. This allows us to see how compilers make use of the relationship between disjoint lexical units called syntax and reasons about them by adding or removing edges.

By now it should be clear that compilers are the key to a self-improving system. They are the key to the ability to reason about the meaning of a program. Now whether complete self-improvement is possible, is a question that remains to be answered. But hopefully we can find an approximate answer to this by the end of this report.

2.3.2 Semantics

60% of the compilation is semantic analysis. So I want to spend some time explaining what this means, and why it is important for our system to become a self-improving system. Semantics can be defined rigorously using the notion of satisfiability and validity. For eg. in the following sentential function, "_____ is a three-volume work on the foundations of mathematics, also considered by many as the first book on Type Theory was written by Alfred North Whitehead and Bertrand Russell", the blank can be satisfied with any word in the dictionary, but the statement can only be called valid if the blank is satisfied with a value that is true. Now, this may not hold for non-factual statements, but for genuine propositional entities, this must always hold true.

In terms of programming, this can be understood using the following example:

- `x = 2`
- `x == 2`

In the first example, we can see that the 2 is being assigned to x, which even in a dynamically typed language could be inferred immediately as an integer, but in the second example, we need to do some kind of type checking to see if the value of x is an integer. This is where the idea of satisfiability and validity comes into the picture. Any expression can satisfy the comparison, but only $x \in Z$ and $x = 2$ can be semantically valid. This is because the expression $x == 2$ is a proposition, and the proposition is true only if x is an integer and x = 2.

To tie it all together, this is the form of reasoning that we want our systems to be able to do, but we want them to understand and come up with novel ways to reason about the meaning of the program so that they can become self-improving.

2.3.3 Consequences

The consequence of having a system, in this case, a compiler, that can reason on its own is that we can outsource a significant amount of High-Performance Work to the compiler. So far the question to compute-intensive tasks has been ”what can we do so that the high-performance systems can complete the task faster”, but this approach to High Performance not only allows us to complete the task faster but also to ask a different question: ”What can the High-Performance Systems do to help us help them complete the task faster?”

At first glance, this question may not seem really interesting or important, but I have come to believe that even a tiny change in perspective can help us completely turn around the way we think about solving a problem, especially if the problem is a challenging one.

3. Compilers and Optimizations

Compilers as we discussed above are epistemic tools that help us reason about the meaning of a program. In technical terms, compilers are rulial[2](see Wolfram) systems that can perform of series of step-by-step transformations on a program. These transformations include things like: Lexical Analysis, Semantic Analysis, and Code Generation. This can be further subdivided into things like Tokenization, Parsing, Type Checking, Type Annotation, Single Static Assignment, IR Generation, Register Allocation, etc.

Modern-day compilers are smart enough to procedurally transform a program such that it is exponentially efficient compared to the original program. This is because compiler design over the three decades has improved tremendously. With the advent of techniques like Single Static Assignment and Value Numbering and Dominance, compilers have become more effective and efficient at target code generation than most human beings.

For example, gcc which is not considered by many as an optimizing compiler is capable of transforming a fairly non-trivial program like `Montgomery Multiplication` to a program that is 25x(shown in Fig 3) as efficient as the original program.

This is a significant improvement over the original program. To accomplish this, modern compilers go through multiple passes of transformation.

I will go through some of the most important phases of compiler construction[1] so that we have some idea about what we mean when we use the term optimization.

3.1 Lexical Analysis

The first pass is called the lexical analysis where the compiler breaks the program into tokens. The tokens are the smallest units of the program.

3.2 Semantic Analysis

The second pass is called the semantic analysis where the compiler analyzes the tokens and determines the meaning of the program. This pass is split into two parts:

- Parsing
- Type Checking

3.2.1 Parsing

Parsing is the process of breaking the program into syntactic units. This is done by the compiler using a set of rules that are defined by the language. The rules are also called the grammar of the language.

```
# gcc -O3

.L0: 3
movq rsi, r9
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rdx, r9
imulq rsi, rdx
imulq rsi, rcx
addq rdx, rax
jae .L2
movabsq 0x100000000, rdx
addq rdx, rcx
.L2:
movq rax, rsi
movq rax, rdx
shrq 32, rsi
salq 32, rdx
addq rsi, rcx
addq r9, rdx
adcq 0, rcx
addq r8, rdx
adcq 0, rcx
addq rdi, rdx
adcq 0, rcx
movq rcx, r8
movq rdx, rdi
```

Figure 3. GCC 03: The optimized Montgomery Multiplication program

The original program with -O0 optimization was 1000 lines long.

```
if (input()):
   x = [1, 2, 3]
else:
   x = 2
do_something(x)
```

Figure 4. The program with type checking

The type of x is determined by the input at runtime. We need type checking to ensure that `do_something` is getting the appropriate type or the compiler should throw an error.

3.2.2 Type Checking

Type checking is the process of determining the type of each token. Usually, type checking and parsing are done simultaneously. There are many ways to do type checking, but one of the most prominent ways is to use something called a type annotation. This is not required in statically typed languages, as the type of each variable is known at compile time. In dynamically typed languages, the type of each variable is inferred at runtime.

For eg. in the following program(See Fig 4)[5], the type of x could be a list or a number depending on the input.

3.2.3 Other Passes during Semantic Analysis

Sometimes the compilers also tend to do things like Closure Conversion and Escape Analysis in this pass. In dynamically typed languages we can have nested functions where the lexical scope of the inner function is not visible to the outer functions. And there are instances where inner functions could be using variables from the outer scope which are not passed as parameters. So to be able to optimize the code and do meaningful analysis on these procedures, we need to lexically bind these variables (Commonly referred to as ”free variables”) so that the ownership can be clearly defined.

3.3 Code Generation

The third pass is called the code generation phase where the compiler generates the machine code for the program.

This part is where we want the reasoning abilities of our compiler to improve so that we can generate more efficient code. So our semantic analysis phase and the first few sub-phases of Code Generation are the most important phases that determine whether we can ever have a self-improving HPC system.

This phase is split into multiple sub-phases, we will not go into details about each sub-phase. But I want to give a brief description(learned from [9] so that we can understand the logic behind superoptimization when we come to it.

3.3.1 Dataflow Analysis[3]

3.3.2 Liveness Analysis

Liveness analysis is the process of determining which variables are alive at each point in the program. This phase can be thought of as a reverse Dataflow phase, where the compiler gathers the data about whether a register or a stack location needs to be assigned to a variable.

Dataflow Equation:

$$\text{LIVE}_{in}[s] = \text{GEN}[s] \cup (\text{LIVE}_{out}[s] - \text{KILL}[s])$$

$$\text{LIVE}_{out}[final] = \emptyset$$

$$\text{LIVE}_{out}[s] = \bigcup_{p \in succ[s]} \text{LIVE}_{in}[p]$$

$$\text{GEN}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{x_1, ..., x_n\}$$

$$\text{KILL}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{y\}$$

This says if the variable is not alive(a.k.a killed) after the current instruction we can assign it a register as it won't interfere with the variables after the current instruction.

3.3.3 Dominance Analysis

Dominance Analysis is a dataflow analysis that determines which computations can be lifted up the Control Flow Graph (CGF). This phase is extremely useful in compiler construction as it allows us to see which variables/expressions can be hoisted out of the loop(where the computation might be performed repeatedly for no good reason, and end up wasting CPU cycles).

$$\text{Dom}(n) = \begin{cases} \{n\} & \text{if } n = n_0 \\ \{n\} \cup \left(\bigcap_{p \in \text{preds}(n)} \text{Dom}(p)\right) & \text{if } n \neq n_0 \end{cases}$$

This equation above tells us how to find the dominators of a basic block in a Control Flow Graph. A control flow graph is a directed graph where each node is a basic block and each edge is an indicator showing the relationship between the basic blocks. And a basic block is a set of instructions that are executed in sequence without any branching or interruption.

Using the above equation we can find the dominators of a basic block and see if we can lift up the computation from that basic block to one of its predecessors. Although doing this is not always possible because of complex control flow, it is a good way to see if we can reduce the number of computations.

```
x1(1) = 1
for i in range(10):
   x1(2) = x1(?) + sum(x1(?))
print(x1(?))
```

Figure 5. A program with SSA Form Conflict

Will the program be semantically correct if we just assign the identifiers 1, 1, and 3 respectively? What about the sum function? Shouldn't the first operand inside the add operation in the loop change to x2 after it is assigned to x1(2)

3.3.4 SSA Form

SSA Form[6][7] (or Single Static Assignment Form) is a part of the dataflow techniques where each variable is assigned a unique identifier. This transformation is especially helpful when we know that the variables are used multiple times in different contexts. This transformation allows us to reduce the usage of memory as every store(A store is a reassignment of a variable) can be allocated a separate memory location or a register owing to the unique store identifier. Now one of the problems with this transformation is that it can create a lot of unnecessary variables. So we need to do some analysis to see if we can eliminate some of the variables. And also this can lead to potential confusion in the code where there is control flow involved. For eg. in the following (See Fig 5), what should we do?

This can be solved using ϕ nodes which are special instructions used to resolve the SSA form conflicts. They use compile-time type checking and give out the correct identifier for the variable. But this requires using advanced transformations such as `Reaching Definition Analysis` and `Dominance Frontier Analysis` to be able to do this.

3.3.5 Reaching Definition Analysis

$$\text{REACH}_{\text{in}}[S] = \bigcup_{p \in pred[S]} \text{REACH}_{\text{out}}[p]$$

$$\text{REACH}_{\text{out}}[S] = \text{GEN}[S] \cup (\text{REACH}_{\text{in}}[S] - \text{KILL}[S])$$

Reaching Definition Analysis is a sub-phase in Data Flow Analysis, which says that a block is reachable from a given block if there is no intervening store(reassignment) from the definition(first assignment). They are used in the context of a Control Flow Graph, where the presence or absence of a store(or the reassignments of a variable) tell us if there is confusion about the variable in question. We can then use this information to resolve the conflicts.

3.3.6 Dominance Frontier Analysis

$\text{DF}(n) =$ First descendent that is not dominated by the block n.

Dominance Frontier Analysis is also a sub-phase in Data Flow Analysis. As mentioned above it is mainly used to decide where to place the Phi nodes. This is because the dominance frontier helps us find the first block that is not dominated by the block we are currently in, so if there is a conflict in the current node (n), we can place the Phi node at DF(n).

Now once we are done with parsing and data flow, the next logical step is to assign the variables a physical home in the processor. And this is where ”register allocation” comes into the picture. It makes use of the previous phases to allocate homes to expressions and variables.

3.4 Register Allocation

Register Allocation as the name suggests is the process of assigning registers to variables. Now with more variables than there are registers, we will have to inevitably spill over some of the variables to the stack or to the heap. This is where efficient register allocation algorithms and dataflow analysis can help us understand which variables really require registers and which can be spilled over. One of the common techniques is to use an interference graph that takes in a set of live variables and analyze which variables interfere with each other and which variables don’t. And based on the ”scope” information from the dataflow analysis and the ”interference” information from the interference graph we can efficiently allocate registers. This mapping of the register to a variable is usually done using some form of graph coloring where each variable is assigned a color corresponding to the register that is most suitable for the variable.

With that said, there are countless other techniques that can be used to allocate registers and perform data flow analysis. Additionally, there are numerous other optimization techniques like vectorization, loop fusion, etc. that can be used to improve the performance of a program. But all of them come with a cost. The cost of universality.

Note: We will not go through these techniques in this report, but we know from experience that they can significantly improve the efficiency of a program.

4. Implications of Deterministic and Universal Optimizations

Now although there are many transformations that can be done on the Program as mentioned above, there are some key issues that still remain. Most of these optimizations are deterministic in nature. This kind of optimization allows for universality but with High-Performance Systems it is not universality that is important. Most of the High-Performance Tasks are extremely parochial in nature. So the optimizations will also have to be done on a per-task basis.

These kinds of optimizations, the deterministic and universal kinds, can give some performance boost, they cannot come up with novel problem-specific optimization. I personally believe that this has made the field of high-performance systems stagnate as a whole.

Entertain this for a moment. What if your compiler could completely get rid of type checking because it understands that your particular program does not use polymorphism? What if your compiler could replace control flow-based instructions with deterministic instructions that are completely novel but somehow preserve the semantics of the original program?

Mind-Boggling, right? We can already do this. Well, to some extent. And guess what, nobody knows about it.

I would have imagined that with the advent of modular compiler infrastructure toolchains like LLVM[11], we would already be injecting our backend/codegen with stochastic optimization techniques that cater to specific programs, but surprisingly enough, this is not the case. My guess is that the idea of ”stochastic = unpredictable” is so deeply entrenched in the minds of the optimizing class that they don't even think that is worth trying out.

I understand it is not fair to paint everyone with the same color, and there is some initiative in the ML community toward this. But this has been a very disappointing story so far. If we wait for humans to have their eureka moments in the domain of compiler optimization we might have to wait another 30 years before we see a breakthrough. On the other hand, some of these systems that I am talking about are already pretty good. They only need to be scaled.

All said, the real question is how would the field of high-performance systems be impacted by this?

And to understand this, we will have to see understand how some of these techniques work and what they can bring to the field of HPC in general.

5. Optimization as a search problem

Optimization problem. Given a set X of candidate solutions, and a criterion map $F : X \to R$, find an $x \in X$ such that $x \in argmax_{y \in X} F(y)$

Optimization is usually seen as an argmax problem where it is a matter of finding the best solution among the candidate solutions while verifying both the domain and the codomain. Whereas a pure search problem only needs verification of the codomain.

One example of this would be the problem of finding semantic validity in terms of action equivalence as opposed to output equivalence. If we see program transformation as an optimization problem, we will find ourselves in the local optima of having to choose between different representations of the same instruction. For eg, Multiplication could be reduced to shift operations, but a fairly non-trivial "if condition" cannot be converted to a combination of deterministic instructions. This requires novel optimizations, and more importantly program-specific parochial optimizations.

My foray into stochastic optimization has me convinced that this is a problem that is not solvable by a deterministic algorithm. With the latter(stochastic method), the search space becomes the entire computational universe i.e., the space of all possible programs that can give the same output as the original program. But with the former, the search space is the set of all possible programs that look more or less the same as the original program but improves upon it by cutting costs on redundant computations.

```
signum (x) int x;
{ if(x > 0)
   return 1;
else if(x < 0}
   return -1;
else
   return 0;
}
```

Figure 6. Superoptimizer by Massalin - Original Code

This is where the notion of stochastic optimization comes in.

6. Superoptimizers

6.1 Background

Superoptimizers are generally stochastic in nature, although not all of them. But the kind of superoptimizers that I am trying to talk about today are stochastic in nature. And they provide results that I know for a fact can never be produced by a human being. We see some of the examples below, but let's first start with the background.

The first use of the word "superoptimizer" can be traced back to the year 1987 when H. Massalin presented his paper titled "Superoptimizer: A look at the smallest program"[12]. This was a fantastic paper that discussed the idea of tuning the generated assembly code to get a slightly different program each time and assigning a cost function to evaluate the performance of the generated program. The main drawback however with this paper is that it only works on extremely simple programs. And because of this, no one took it seriously, I'd say even now, but until the year 2012 when Stoke was released. Superoptimizer by Massalin, although small, could generate really novel programs. One example that he shared in his paper, which really took me by surprise was the following (See Fig 6 and Fig 7):

And remember that we are talking about 1987. No, we did not have SSA Form or Complex dataflow analysis back then. What's really surprising about this result is that it in no way resembles the original program. The generated program just by means of trial and error figured out that a carry flag can be used to determine if a number was positive or negative. This is all the more interesting to me because the entire experiment was done

```
(x in d0)
add.l d0,d0
| add d0 to itself
subx.l dl,dl
| subtract (dl + Carry) from dl
negx.l d0
| put (0 - d0 - Carry) into d0
addx.l dl,dl
|add (dl + Carry) to dl
(signum(x) in dl}
| (4 instructions}
```

Figure 7. Superoptimizer by Massalin - Optimized Code

on a 16MHz 68020 computer. Massalin mentions in his paper that superoptimizers have little use in code generation but little did he know about the indirect benefits of Moore's law and the magic of applied mathematics.

6.2 Souper

Souper[13], an LLVM-based synthesizing superoptimizer does exactly this. It assists in code generation by creating novel IR (intermediate representations). Souper works by creating a new IR based on the IR generated by LLVM for each block. It creates a DAG of rhs expressions of each instruction in the block and then synthesizes new instructions for each of the nodes using stochastic techniques. Upon successfully synthesizing new instructions, souper uses an equivalence function to verify if the RHS evaluates to the same value that the LHS holds in the original program. Once verified the LHS and the new RHS are then concatenated to form a new instruction.

Souper works only on control-flow-free programs, but it can be used on a program with control flow using something called blockpc. A blockpc is a llvm construct(an analog in computer architecture would a program counter) that is used to keep track of the current block. This is very useful in the case of loops and if statements as it allows souper to incorporate the information from data flow analysis into the synthesis.

The essence of synthesis is finding a cost-minimizing solution to an exist-forall formula. In other words, when we talk about stochastic techniques we want to prove that there exists a way to connect up a collection of instructions such that, for all inputs, the resulting RHS behaves the same as its LHS, and furthermore, the synthesized RHS is the cheapest possible RHS. As discussed above, souper uses an equivalence function to verify the satisfiability, but for search and cost analysis, it uses a counter-example guided inductive synthesis (CEGIS) algorithm developed by Gulwani et al[10]. This CEGIS based algorithm avoids exhaustive search and also avoids producing queries that contain nested quantifiers. Rather, given a collection of instructions, it formulates a query permitting all possible producer-consumer relationships between the instructions, with the position of each instruction being represented as a line number. This query is satisfiable if there exists a way to connect the instructions into an RHS that is equivalent to the LHS for just the satisfying input. If this is the case, the instructions and the constants in the RHS can be reconstructed from the model provided by the solver in the equivalence function. Then, in a second step, CEGIS asks the solver if the straight-line program on the RHS is equivalent to the LHS on all possible inputs. If so, synthesis has succeeded. If not, constraints are added to prevent the solver from traversing this particular circuit a second time, and the process is repeated. In doing this process repeatedly, Souper uses measures such as the number of instructions, type of instructions, instruction width, etc to determine if the argmax is attained or not.

Below is an example program(See Firg 8, 9, and 10) where Souper converts the control flow-based program to a program that only consists of blockpc instructions, and synthesizes the result well before the target code generation phase.

Although impressive in its ability to synthesize programs, Souper is not without its flaws. For example, there have been instances of miscompilations when running souper along with other LLVM passes such as Global Value Numbering, Instruction Merging, etc. Especially when the behavior is not strictly defined, souper has a hard time figuring out what to do.

Also because it uses an SAT solver as an equivalence function, if the solver says that the expressions are equivalent when they really aren't, Souper will yield a wrong answer. This is to say that soundness checking is not one of Souper's strengths.

6.3 Stoke with Sound Loop Superoptimzer

This leads us to our next Superoptimizer also, coincidentally, titled the Sound Loop Superoptimizer. Sound Loop Optimizer[8] is a Stoke-based superoptimizer

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

Figure 8. C Program with Control Flow

that is able to incorporate robust soundness checking into the synthesis process. But before I go into Sound Loop Optimizer, I want to talk about Stoke, so that we can understand Sound Loop Optimizer better.

6.4 Stoke

Stoke[14] is a stochastic superoptimizer developed by a group of researchers at Stanford. The main idea behind Stoke is to use the random walk to walk through the search space and use a cost function comprising equivalence and performance to guide the search. As mentioned above, Stoke is the first superoptimizer since the original paper Massalin, that took the idea of stochastic optimization seriously. Stoke was designed to work on real-world programs. And as we will see in the following sections, Stoke outperformed Gcc -03 and Clang -03 by producing code that was 5x faster than the ones generated by the aforementioned compilers.

The key idea behind superoptimization is to perturb the program slightly by adding an instruction, removing an instruction, or fiddling with the operands, and checking if the resulting program is sound. Stoke in order to accomplish this, uses the notion of rewrites. A rewrite(R) is when a given string is transformed into a new string by means of insertion, deletion, or replacement of instructions.

Stoke uses a markov chain monte carlo method based algorithm to walk through the search space and find/synthesize a rewrite. And to constrain and validate,

```
unsigned g(unsigned a)
define i32 @g(i32 %0)
{
  label %1:
  %2 = urem i32 %0, 4
  switch i32 %2, label %9 [
    i32 0, label %3
    i32 1, label %5
    i32 2, label %7
  ]
  label %3:
  %4 = add i32 %0, 3
  br label %9
  label %5:
  %6 = add i32 %0, 2
  br label %9
  label %7:
  %8 = add i32 %0, 1
  br label %9
  label %9:
  %.0 = phi i32
  [ %0, %1 ],
  [ %8, %7 ],
  [ %6, %5 ],
  [ %4, %3 ]
  %10 = and i32 %.0, 3
  ret i32 %10
}
```

Figure 9. LLVM IR with labels and phi nodes

it uses a cost function comprising of an equivalence function given by

$$Eq(R, R') = SAT(R, R')$$

and a performance function given by

$$Perf(R, R') = InstCount(R, R') \cap BitWidth(R, R')$$

to get the optimal rewrite.

$$C(R; T) : Eq(R; T) + Perf(R; T)$$

For eg, if you recall from our earlier discussion on compiler optimization where we talked about gcc -03 output on Montgomery Multiplication. Stoke was able to synthesize a 8x faster target version of the Montgomery Multiplication (See Fig 11).

```
%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
```

Figure 10. Souper IR with no labels

We can see how Souper is able to convert a fairly complex control flow based program into a program that only consists of blockpc instructions. And moreover, it synthesizes a single instruction in its subsequent pass that is equivalent to the original program.

Now the problem is that stoke shares the same set of flaws that Souper has. The only way it can check soundness is by using an SAT solver and manual test cases.

This is where the idea of a Sound Loop Optimization comes into the picture.

6.5 Sound Loop Optimization

Most superoptimizers verify the soundness of the synthesized program by using an SAT solver and manual test cases[15]. But it should be obvious to anyone who has any amount of programming experience that corner cases are a thing. And no amount of manually generated test cases can be considered exhaustive enough to give the synthesized program a straight A. So, a team of developers from Google and the core team of Stoke set out to improve the soundness aspects of superoptimizers.

```
# STOKE

.L0:
shlq 32, rcx
movl edx, edx
xorq rdx, rcx
movq rcx, rax
mulq rsi
addq r8, rdi
adcq 0, rdx
addq rdi, rax
adcq 0, rdx
movq rdx, r8
movq rax, rdi
```

Figure 11. STOKE: x86 repr of Montgomery Multiplication

If you go back to Fig 3, you can see the stark difference between the two versions. Stoke was not only able to synthesize a faster version, but it was also able to introduce novel logic without losing the semantics of the original program.

The reason that this paper was interesting to me was that the only thing stopping the wide adoption of superoptimzers is its inability to guarantee formal correctness and soundness.

And Sound loop optimization makes use of two-fold techniques called ”Bounded Verifier with Automatic test case Generator and Sound Verifier” to verify the soundness of the synthesized program.

A bounded verifier performs a partial proof of equivalence for a user-supplied parameter k, it checks whether the target t and the rewrite r agree on all inputs for which every loop in each program executes for at most k iterations. If the check fails then the bounded verification produces a counter-example demonstrating the difference. Otherwise, the programs are equivalent up to the bound K and the differences (if any) can only be demonstrated by running the bounded verifier on a larger bound K. In contrast, the sound verifier performs a sound proof of equivalence by mining the rewrite aliases and seeing if the semantics has changed or not since the last rewrite.

Although it is still not perfect or exhaustive by any means, with this approach Sound Loop Optimizer is able to get the benefit of both worlds, and produce

```
(sub X
  (shl X, 1))
```

Figure 12. Web Assembly using SExpression syntax

efficient and sound code that is also exponentially faster than the generically optimized code.

7. Implications of Superoptimizers

Imagine that you have a program that looks like the one in Fig 12. Now most modern-day optimizing compilers will not be able to optimize the program any further. But with superoptimizers, we can figure out that the program is just doubling and halving the value of the variable. Stoke, for example, was able to generate `X` as output for the program in Fig 12. And this is a very trivial program.

This is to say that High-Performance software systems work on computationally intensive tasks all the time, but we rely on generic optimizations for the most part. So, if we can figure out a way to incorporate stochastic optimization techniques into our high-performance software development workflow, particularly in the compilers that are used in compiling these high-performance software, we can exponentially increase the performance of our programs and all without incurring any additional cost.

With the advent of modular compiler infrastructure toolchains like LLVM, this has only become easier to do. And with the versatility of LLVM IR, we can easily incorporate stochastic optimization techniques in the backend or in the middle-end without having to worry about the soundness or the performance.

My hope is that one day we as a community eventually move towards a place where we can marry machines together, not only to use each other's resources for performing user-level tasks, but also for talking to each other and improving themselves so that we can all be more efficient and effective.

I don't know if we can ever objectively disprove Amdahl's law, but I definitely think that we can disregard Amdahl's law in the future if we move towards a more modularized, more stochastic world of programming. As a rational optimist, I think that novelty and creativity are the keys to the future of programming. Note that I am not saying that humans cannot be creative, but that the human mind needs the right kind of tools to do what it does best. And hopefully, stochastic techniques can play a role in that process.

8. Future Work

There is plenty of work that is still left to be done. Although great in terms of synthesis, the superoptimizer is still not ready for production use. Real-time mission-critical systems cannot gamble on the quality of generated code. So, we need to think about how to make the superoptimizers more robust. We also need to look into how we can expand the use case of superoptimizers in domains other than compilers. I believe that most programming/software development is just a Postian[4](See Emil Post) string rewrite problem, and if we can somehow find a cost function that can constrain the synthesis process, superoptimizers can be used in more domains.

Going forward, I want to develop a prototype superoptimizer using modern Deep Learning techniques for Web Assembly. I want to see if we can improve the state of web programming in general. Modern-day web programming suffers from all sorts of maladies owing to its tight coupling with Javascript. And with major corporations using Javascript for the majority of their user-facing applications, it is very important that we find a way to enhance the performance of web programming in general. And if we can stretch the boundaries of HPC to include the work of organizations like AWS and Google cloud as High-Performance Computing, we can see why improving the state of the web would be a significant step in the right direction.

9. Conclusion

"Topics on HPC" was one of the most rewarding courses I've taken at CU. This course not only allowed me to dig deeper into one of my favorite subfields of computer science but also allowed me to intertwine my interests in compilers with my interest in HPC. Before this course, I was only focused on the user side of compiler optimization. Researching for this course allowed me to understand the inner workings of the compilers in great detail. Additionally, since the course was on high performance, this forced me to find an intersecting area between the two fields — Superoptimization. This has been a revelation of sorts for me. I would have never, in a million lifetimes, thought that automatically adding or removing random strings from/to a program would be a good idea. But researching for this course has proven my skepticism wrong.

Initially, I was very interested in developing a toy implementation to demo for my final presentation. However, to get a working demo of a superoptimizer would have been a fairly large endeavor. So I went deep into the implementational aspects of multiple superoptimizers. I can only hope that everyone in the class found it as exciting as I did. That said, I intend to take the learnings from this course and apply them to my own work in the future.

References

[1] Compilers: Principles, Techniques, and Tools (Dragon Book). URL: `https://suif.stanford.edu/dragonbook/`.

[2] The Concept of the Ruliad—Stephen Wolfram Writings. URL: `https://writings.stephenwolfram.com/2021/11/the-concept-of-the-ruliad/`.

[3] DATAFLOW ANALYSIS. URL: `https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html`.

[4] Emil Leon Post. URL: `https://en.wikipedia.org/w/index.php?title=Emil_Leon_Post&oldid=1073523695`.

[5] Essentials of Compilation. URL: `https://github.com/IUCompilerCourse/Essentials-of-Compilation`.

[6] SSA (GNU Compiler Collection (GCC) Internals). URL: `https://gcc.gnu.org/onlinedocs/gccint/SSA.html`.

[7] Understanding static single assignment forms. URL: `https://blog.yossarian.net/2020/10/23/Understanding-static-single-assignment-forms`.

[8] Berkeley Churchill, Rahul Sharma, and JF Bastien. Sound Loop Superoptimization for Google Native Client. page 14.

[9] Keith Cooper. Praise for Engineering a Compiler. page 833.

[10] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. page 12.

[11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE. URL: `http://ieeexplore.ieee.org/document/1281665/`, `doi:10.1109/CGO.2004.1281665`.

[12] Henry Massalin. Superoptimizer – A Look at the Smallest Program. page 5.

[13] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. URL: `http://arxiv.org/abs/1711.04422`, `arXiv:1711.04422`.

[14] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. page 11.

[15] Rahul Sharma, Eric Schkufza, and Alex Aiken. Conditionally Correct Superoptimization. page 16.