Final Paper
for
CSC 431

[redacted]

[redacted]

Contents

1 Overview

I chose initially to write my compiler in Java. But after learning more Scala to work on my project for [redacted], I chose to use Scala for the later parts (optimization, code generation, and register allocation).

1.1 Parsing

The compiler creates an Antlr `CharStream` from the input file. This is then passed to the generated lexer, and the resulting token stream is passed to the generated parser. The success or failure of the parse is represented with FunctionalJava's `Option` type, with a failed parse resulting in `None` and a successful parse resulting in a `Some` containing the parse tree. Since the parser handles printing parse errors, there is no need to store the parse error in the result structure.

1.2 Static Semantics

An interface for all program validations was defined and used to provide a uniform way to run them. This allows validations to be added or removed as desired, making the compiler more modular.

Return statement checking was implemented as a program validation. It ensures each function has a return-equivalent statement in it. A return-equivalent statement is defined as a block containing a return-equivalent statement, a conditional statement in which both branches contain a return-equivalent statement, a recursive invocation of the function, or an actual return statement. With those definitions, checking a function for a return-equivalent statement requires only a simple recursive function.

Typechecking was implemented as another program validation. An interface named `Typed` was created and all type-checkable AST elements implement it. This interface provides a single method, `type()`, which returns a FunctionalJava `Validation`[1] containing either an error (of type `ValidationException`) or the type of the element it was called on, represented as one of the provided types. For statements, `type()` will return `VoidType` if the types for any subexpressions are valid. Thus, to typecheck a function, we call the `type()` method on its block statement and check if the result is successful or not (i.e., if the result is a failure or an instance of `VoidType`).

1.3 Intermediate Representation

As the result of parsing and validation, an abstract syntax tree is produced, representing the program in a language-dependent fashion. The compiler converts this to a list of functions, each represented as a control flow graph, with LLVM IR instructions stored in each block of the graph. The LLVM instructions provide a layer of abstraction between the code and the hardware instruction set, while not being dependent on the language's structure. LLVM IR is also designed to be in Static Single Assignment form. This makes some program transformations easier and more efficient, as it stores the use-def chain as part of the IR[1]. Converting to a CFG reorients the structure of the program code around the control flow, rather than the syntax of the source language[2]. This also helps to abstract the program representation into a unified shape for all source languages.

In the compiler presented here, Derive4J was used to generate classes and a Visitor pattern implementation in Java for the instructions in the IR. The rest of the classes for the CFG are just plain Java classes. The CFG basic block class stores links to its child blocks to create the graph, but it is not necessary to traverse these links to visit all blocks, as blocks are also stored in a list attached to the function they are from. This makes printing them out much simpler.

Derive4J worked reasonably well, although its generated pattern matching was less flexible than Scala's. I would probably choose to use Scala case classes if I were to reimplement this part of the code.

1.4 Optimizations

Two optimizations were implemented for this compiler: constant propagation and useless code elimination.

To implement constant propagation, the Sparse Conditional Constant Propagation algorithm was used. This algorithm tracks the blocks that are able to be executed, given what is known about the constant values in the program. For example, if a conditional branch has a constant value for a condition, then the algorithm will ignore the block that the untaken branch would have jumped to (unless, of course, it is used by another branch statement). Within the blocks that *are* executed, each instruction is abstractly executed in order

[1] A `Validation` is an `Either` designed for failure-checking. Thus, it provides some convenience methods for accumulating failures across a list of `Validations`.

to determine its result value. This is effectively interpretation of the instructions, but with the ability to handle cases where the actual input values of the instructions are unknown or unknowable.

Useless code elimination is very straightforward: instructions with unused results are eliminated, except for those with possible side-effects (i.e., calls).

1.5 Code Generation and Register Allocation

The SSA code generated by the compiler up to this point contains phi nodes which provide the logic to enable separate execution paths to join into one. Phi nodes do not represent an actual instruction; rather, a single register can be written to by any possible execution path, and the result will always be present for the next instruction. In order to convert phi nodes to plain instructions, it is necessary to substitute a register that will hold the result value from whichever branch executes. A new virtual register is generated to hold the phi value between the parent basic blocks and the block containing the phi node. New `mov` instructions are then added to the parent blocks immediately above the branch blocks ending them. Finally, the phi node is replaced with a `mov` instruction copying the value from the new temporary register to the target register of the phi instruction.

It was noticed that the resulting code sometimes would load a function argument into a register, only to move it to `r0` for the call. In order to avoid this, a minor optimization was added to the register allocator to attempt to reduce all such needless `mov` operations. When doing the graph coloring, the allocator is passed a mapping from registers to registers they are moved to. This is then checked for each register, and if a destination register is not in use, it will be used instead of choosing a new one. This allows the `mov` operation to then be eliminated in a later minor optimization.

This unneeded `mov` elimination optimization occurs as the code of the program is being written into the output file, by ignoring unneeded `mov`s.

One noteworthy thing about the implementation of the stack-based translation is that it makes no attempt to be clever about the loads and stores involved: it potentially will spill a register containing a stack address to the stack, rather than throwing it away and recalculating it when needed.

1.6 Other

In [redacted] this quarter, my team and I built [redacted] We chose to use Scala to do so, as [redacted] had some experience with it. As I became more familiar with Scala, I found I rather liked it, so I transitioned this project over to it. Thus, the early parts of the project are written in Java, typically using the Functional Java and Derive4J libraries, and the later parts are written in Scala.

Also, for amusement, I chose to use a more functional style in many places. I believe this is my first large project in such a style, so there are a lot of icky parts, as I don't really have a good feel for how to implement things cleanly yet.

The translation to ARM assembly is mostly working; I hope that the main remaining bug is a failure to add loop bodies as their own children. As far as I can tell, this leads to the register allocator deciding it can overwrite the registers that are needed by the loop during the first loop run, resulting in complete nonsense.

2 Analysis

These results were run on a Intel Core i7-2600K CPU running in frequency autoscaling mode, with approximately 11 Gb of free memory. Some other programs were running, but system load is believed to have been below 1. The OS is Ubuntu 16.04 LTS. LLVM 3.8 was targeted, as it was available on the testing machine. Gradle 3.4.1 running on Java 1.8 was used to build the code (and it is believed the same JVM was used to run it). A Ruby script was used to gather the reported data. It uses the `benchmark` library from the Ruby standard library to collect the timing data. The times reported are wall clock times. The graphs below are based on ten runs of the program generated by each configuration. The instruction counts are reported by the compiler in a file named `compileStats` which is overwritten after each compilation.

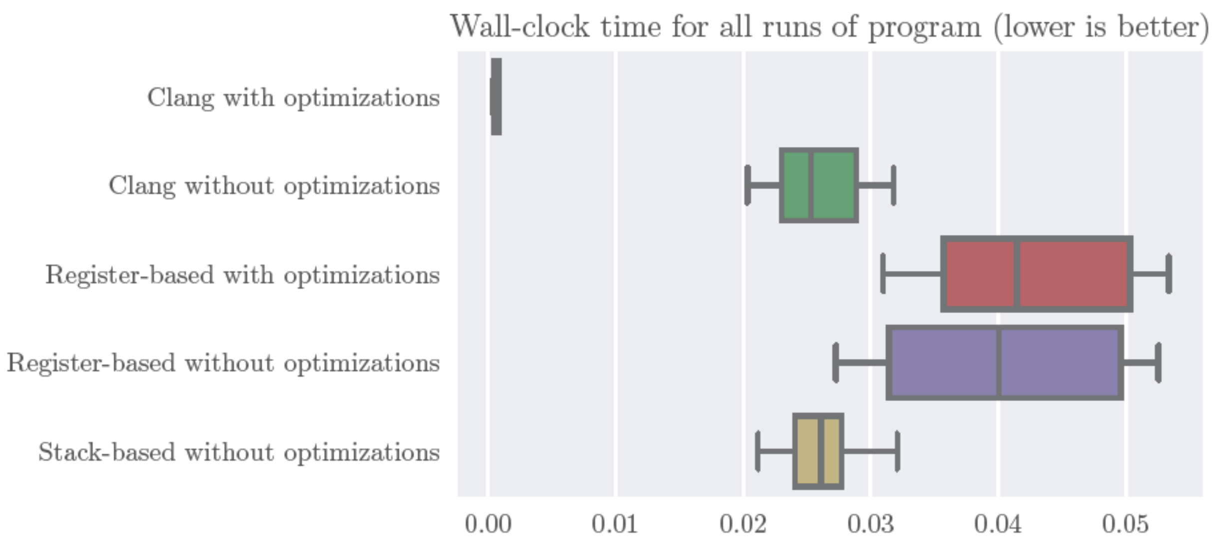
As previously noted in class, the LLVM results are uneven at best. Unsurprisingly, Clang generally beats everything else when all the optimizations are enabled. But the optimizations, and even the use of stack- versus register-based code, do not consistently improve performance. This is likely due to Clang's inability to use smart register allocation without enabling optimizations, and thus the LLVM IR ends up using the stack anyway.

It does appear that in general the register-based code is significantly shorter, with an average reduction in length of 46.53%

2.1 OptimizationBenchmark

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	2.621×10^{-2}	3.376×10^{-3}
Register-based without optimizations	10	4.020×10^{-2}	9.788×10^{-3}
Register-based with optimizations	10	4.248×10^{-2}	8.491×10^{-3}
Clang without optimizations	10	2.584×10^{-2}	4.086×10^{-3}
Clang with optimizations	10	6.089×10^{-4}	1.944×10^{-4}

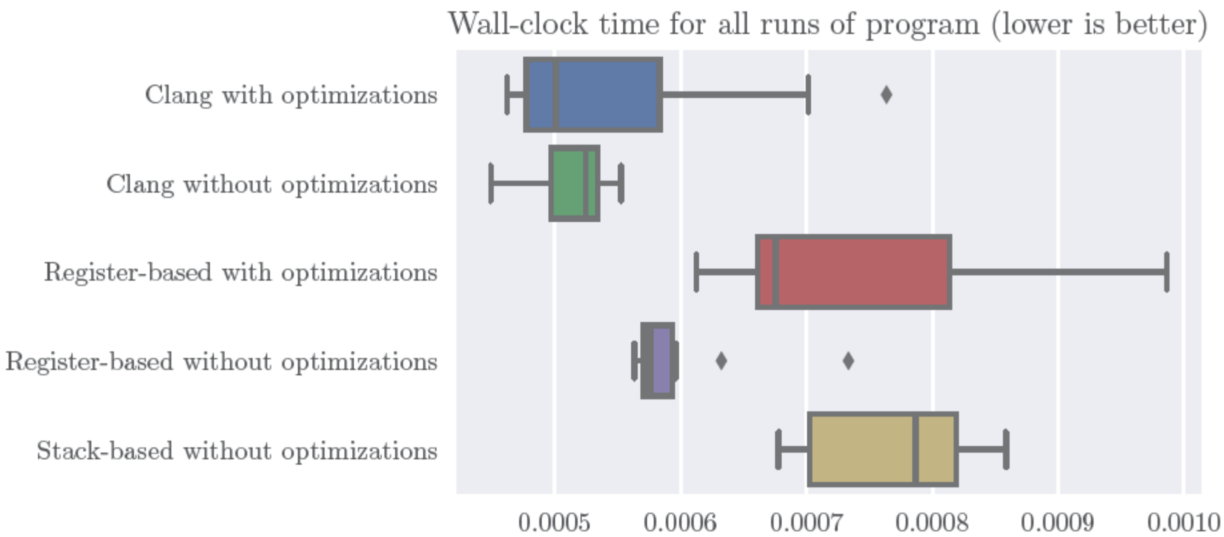
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	974	
Register-based without optimizations	369	62.11
Register-based with optimizations	137	62.87



2.2 BenchMarkishTopics

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	7.687×10^{-4}	6.722×10^{-5}
Register-based without optimizations	10	5.973×10^{-4}	5.188×10^{-5}
Register-based with optimizations	10	7.390×10^{-4}	1.246×10^{-4}
Clang without optimizations	10	5.118×10^{-4}	3.574×10^{-5}
Clang with optimizations	10	5.488×10^{-4}	1.059×10^{-4}

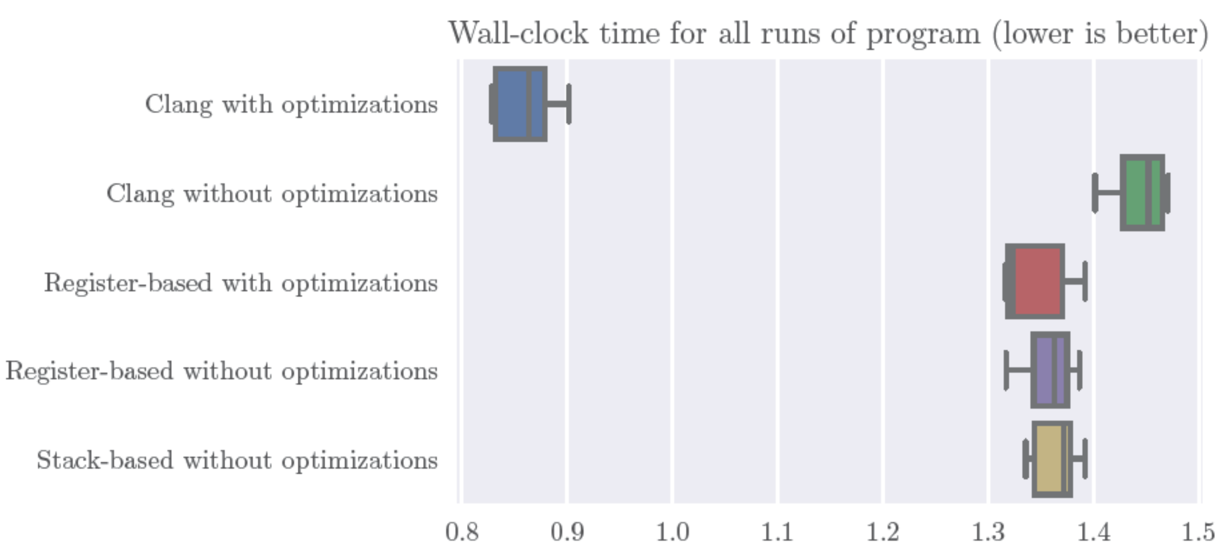
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	113	
Register-based without optimizations	65	42.48
Register-based with optimizations	65	0.



2.3 Fibonacci

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.365	2.098×10^{-2}
Register-based without optimizations	10	1.358	2.267×10^{-2}
Register-based with optimizations	10	1.343	3.103×10^{-2}
Clang without optimizations	10	1.444	2.678×10^{-2}
Clang with optimizations	10	8.591×10^{-1}	2.740×10^{-2}

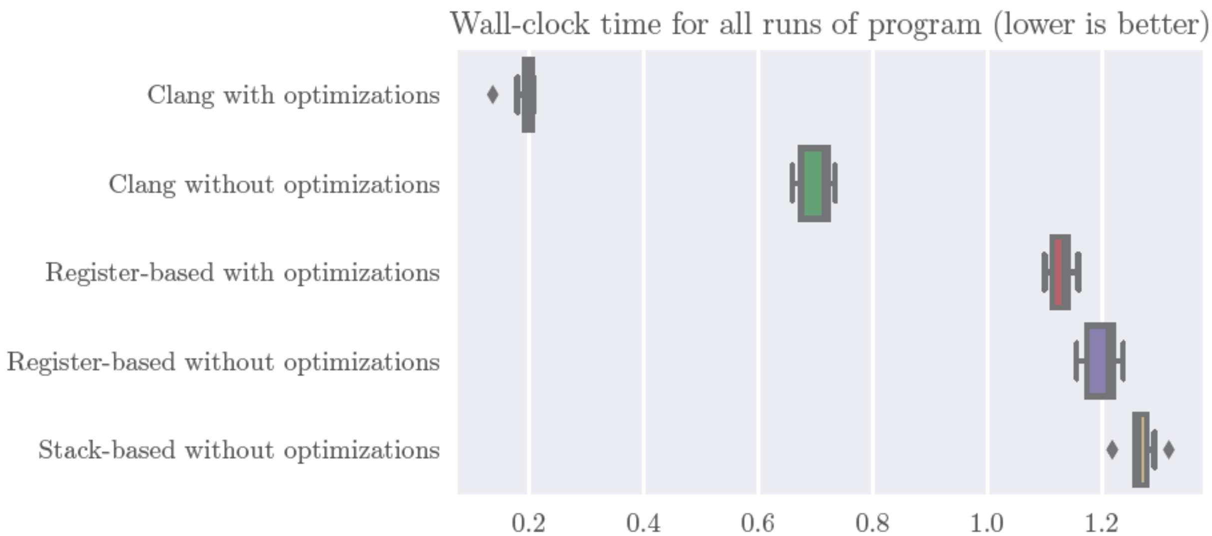
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	26	
Register-based without optimizations	17	34.62
Register-based with optimizations	17	0.



2.4 GeneralFunctAndOptimize

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.265	3.027×10^{-2}
Register-based without optimizations	10	1.200	3.027×10^{-2}
Register-based with optimizations	10	1.129	1.931×10^{-2}
Clang without optimizations	10	7.036×10^{-1}	2.975×10^{-2}
Clang with optimizations	10	1.936×10^{-1}	2.161×10^{-2}

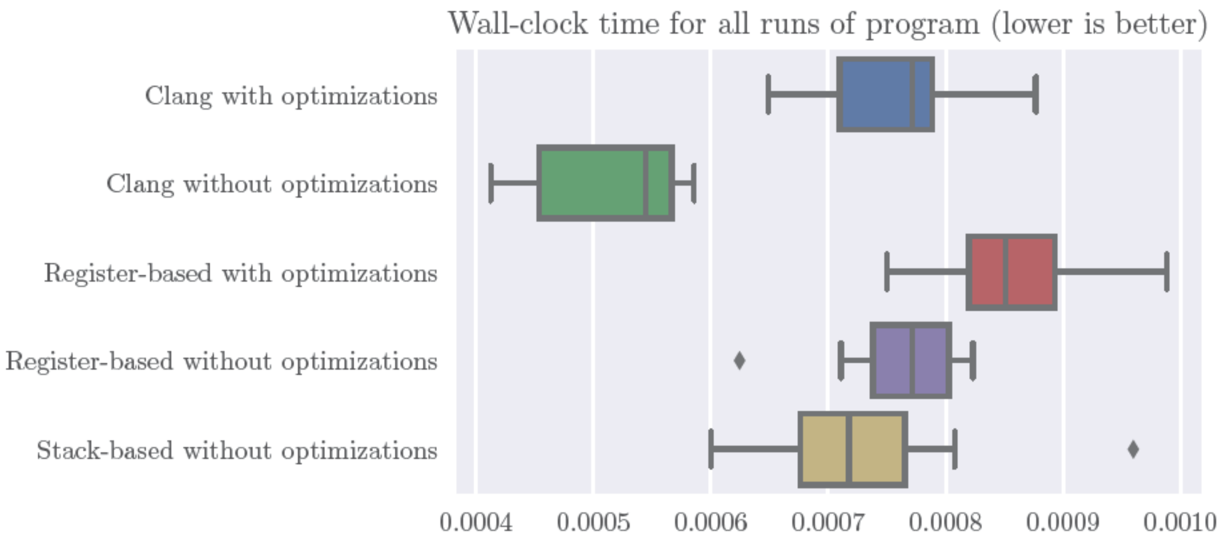
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	141	
Register-based without optimizations	81	42.55
Register-based with optimizations	67	17.28



2.5 TicTac

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	7.322×10^{-4}	1.017×10^{-4}
Register-based without optimizations	10	7.606×10^{-4}	6.114×10^{-5}
Register-based with optimizations	10	8.551×10^{-4}	7.489×10^{-5}
Clang without optimizations	10	5.138×10^{-4}	6.619×10^{-5}
Clang with optimizations	10	7.559×10^{-4}	6.792×10^{-5}

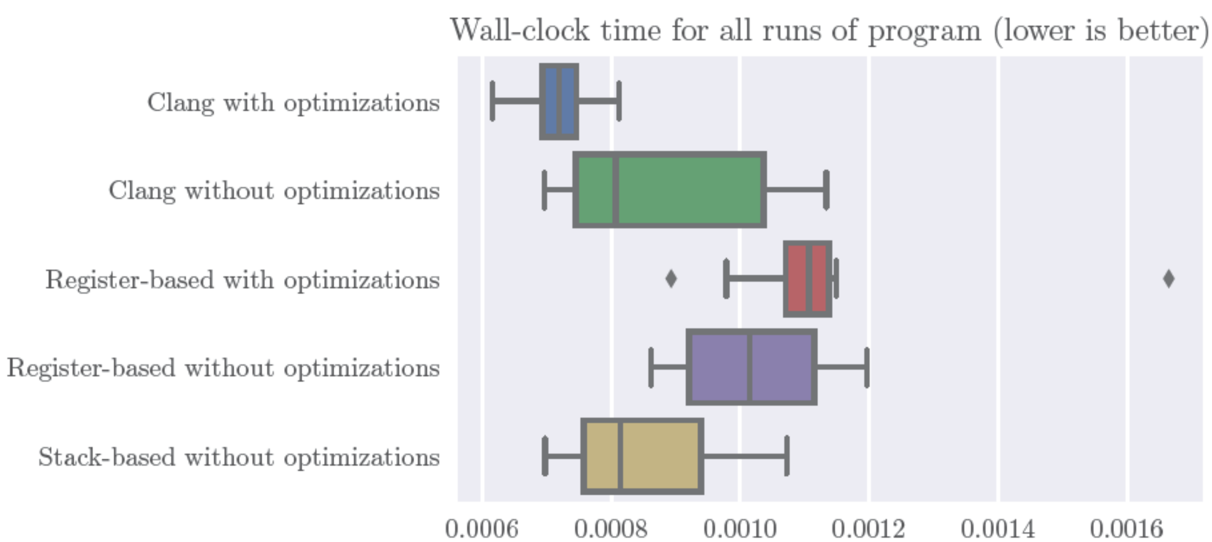
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	469	
Register-based without optimizations	342	27.08
Register-based with optimizations	337	1.462



2.6 bert

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	8.452×10^{-4}	1.225×10^{-4}
Register-based without optimizations	10	1.026×10^{-3}	1.226×10^{-4}
Register-based with optimizations	10	1.132×10^{-3}	2.037×10^{-4}
Clang without optimizations	10	8.867×10^{-4}	1.712×10^{-4}
Clang with optimizations	10	7.214×10^{-4}	5.677×10^{-5}

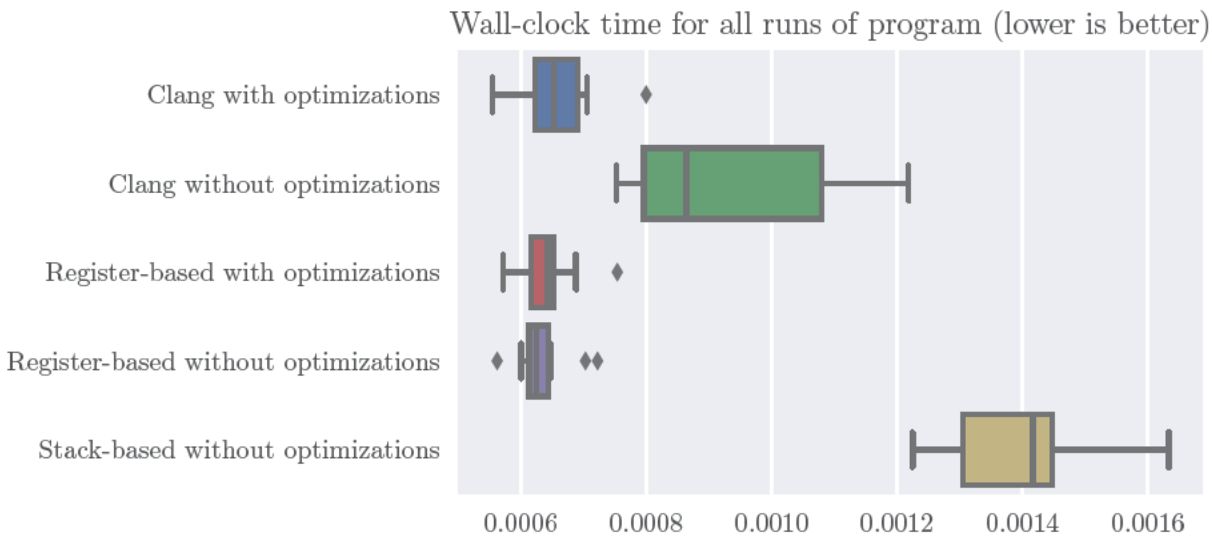
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	617	
Register-based without optimizations	359	41.82
Register-based with optimizations	334	6.964



2.7 biggest

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.397×10^{-3}	1.257×10^{-4}
Register-based without optimizations	10	6.346×10^{-4}	4.718×10^{-5}
Register-based with optimizations	10	6.443×10^{-4}	5.013×10^{-5}
Clang without optimizations	10	9.310×10^{-4}	1.876×10^{-4}
Clang with optimizations	10	6.578×10^{-4}	6.883×10^{-5}

Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	79	
Register-based without optimizations	42	46.84
Register-based with optimizations	41	2.381

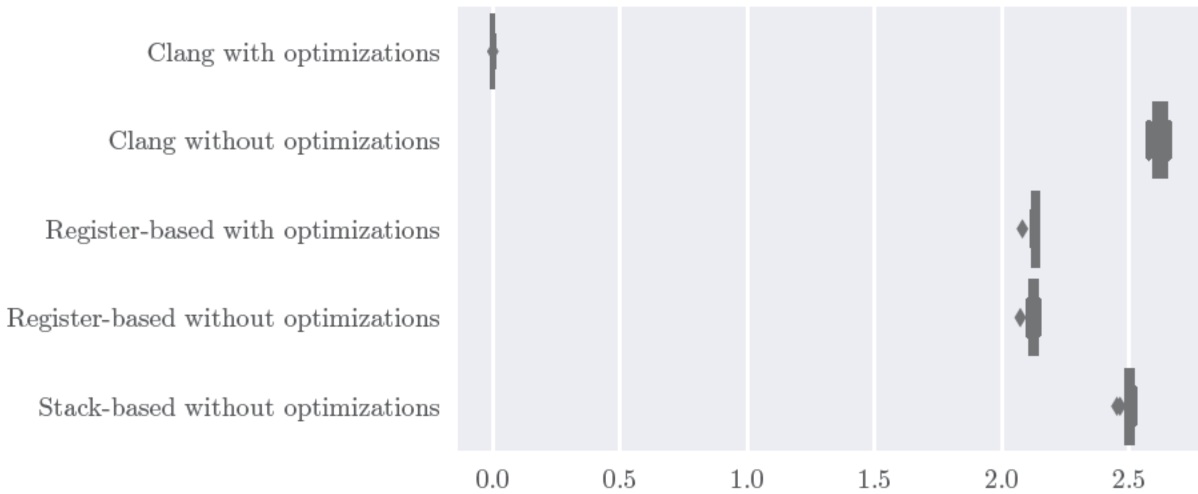


2.8 binaryConverter

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	2.499	2.252×10^{-2}
Register-based without optimizations	10	2.122	2.088×10^{-2}
Register-based with optimizations	10	2.127	1.743×10^{-2}
Clang without optimizations	10	2.623	2.636×10^{-2}
Clang with optimizations	10	7.595×10^{-4}	5.312×10^{-5}

Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	113	
Register-based without optimizations	43	61.95
Register-based with optimizations	43	0.

Wall-clock time for all runs of program (lower is better)

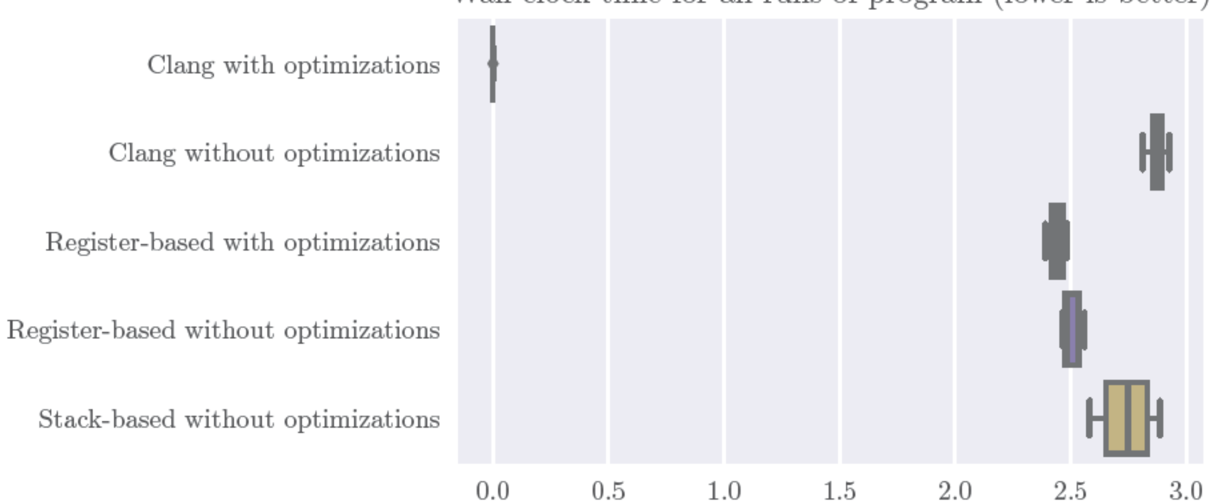


2.9 creativeBenchMarkName

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	2.738	1.161×10^{-1}
Register-based without optimizations	10	2.503	3.726×10^{-2}
Register-based with optimizations	10	2.442	3.451×10^{-2}
Clang without optimizations	10	2.873	3.374×10^{-2}
Clang with optimizations	10	6.279×10^{-4}	7.064×10^{-5}

Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	230	
Register-based without optimizations	116	49.57
Register-based with optimizations	113	2.586

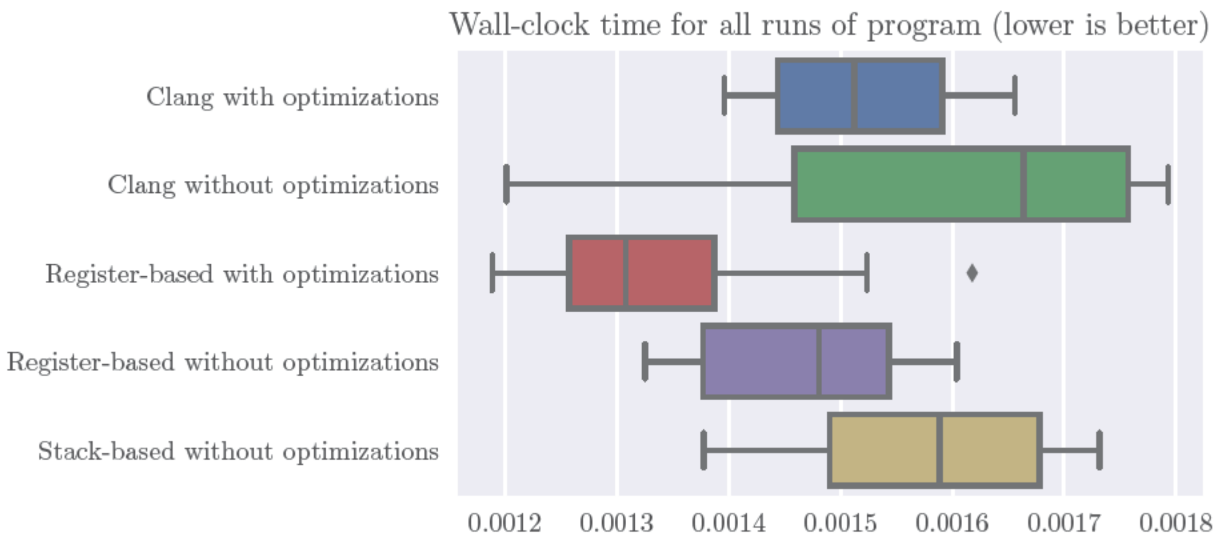
Wall-clock time for all runs of program (lower is better)



2.10 fact_sum

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.576×10^{-3}	1.260×10^{-4}
Register-based without optimizations	10	1.463×10^{-3}	9.963×10^{-5}
Register-based with optimizations	10	1.346×10^{-3}	1.340×10^{-4}
Clang without optimizations	10	1.603×10^{-3}	1.999×10^{-4}
Clang with optimizations	10	1.514×10^{-3}	9.361×10^{-5}

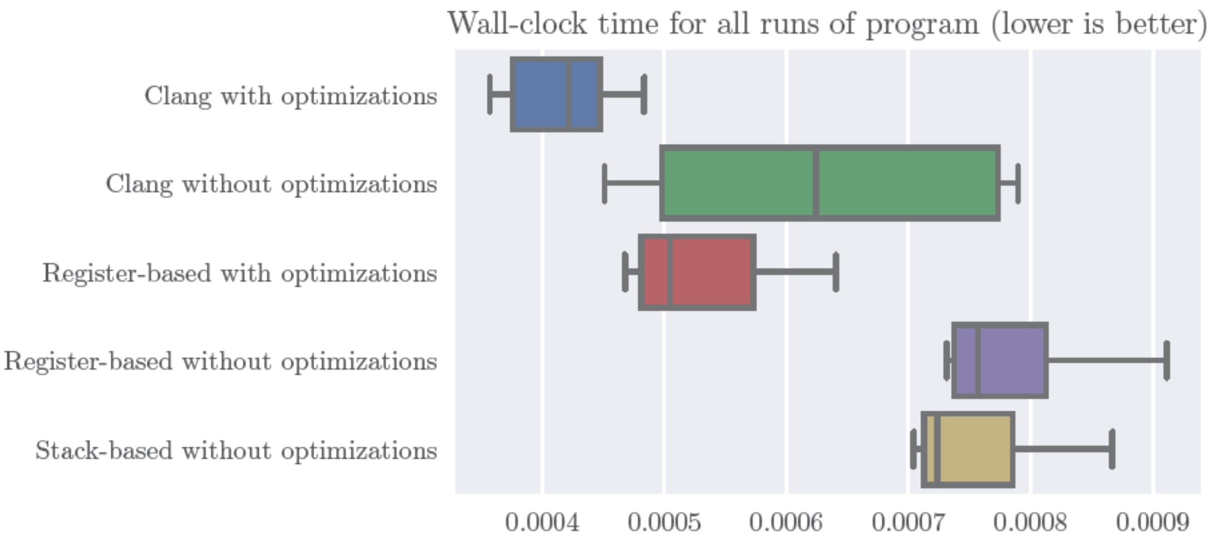
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	66	
Register-based without optimizations	34	48.48
Register-based with optimizations	30	11.76



2.11 hailstone

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	7.530×10^{-4}	5.787×10^{-5}
Register-based without optimizations	10	7.821×10^{-4}	6.118×10^{-5}
Register-based with optimizations	10	5.299×10^{-4}	6.103×10^{-5}
Clang without optimizations	10	6.275×10^{-4}	1.512×10^{-4}
Clang with optimizations	10	4.154×10^{-4}	4.345×10^{-5}

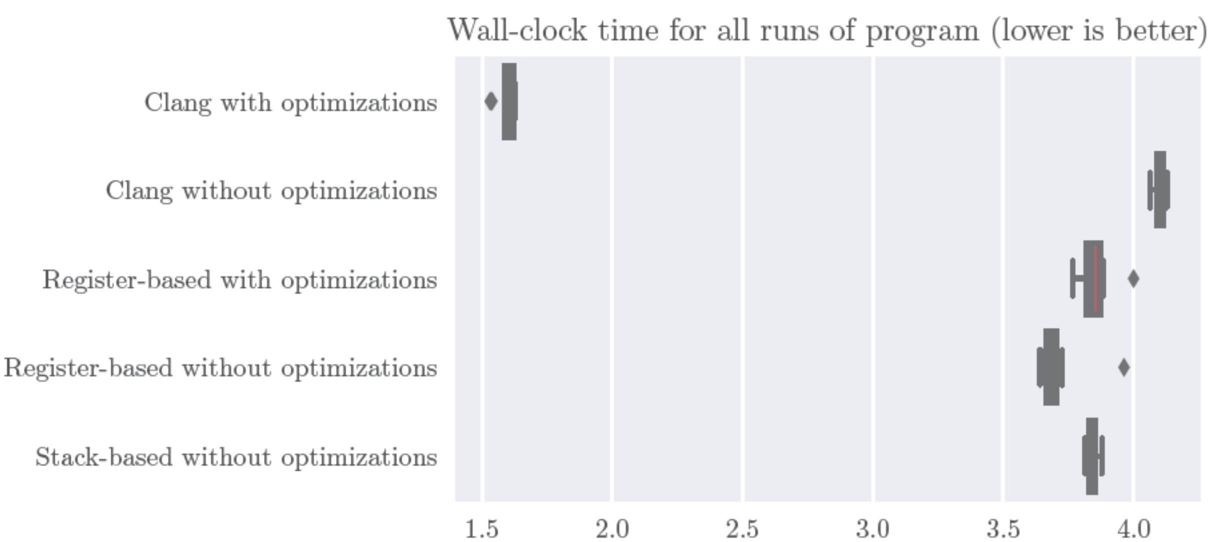
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	45	
Register-based without optimizations	24	46.67
Register-based with optimizations	23	4.167



2.12 hanoi_benchmark

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	3.840	1.918×10^{-2}
Register-based without optimizations	10	3.708	9.307×10^{-2}
Register-based with optimizations	10	3.851	6.220×10^{-2}
Clang without optimizations	10	4.100	1.789×10^{-2}
Clang with optimizations	10	1.593	3.334×10^{-2}

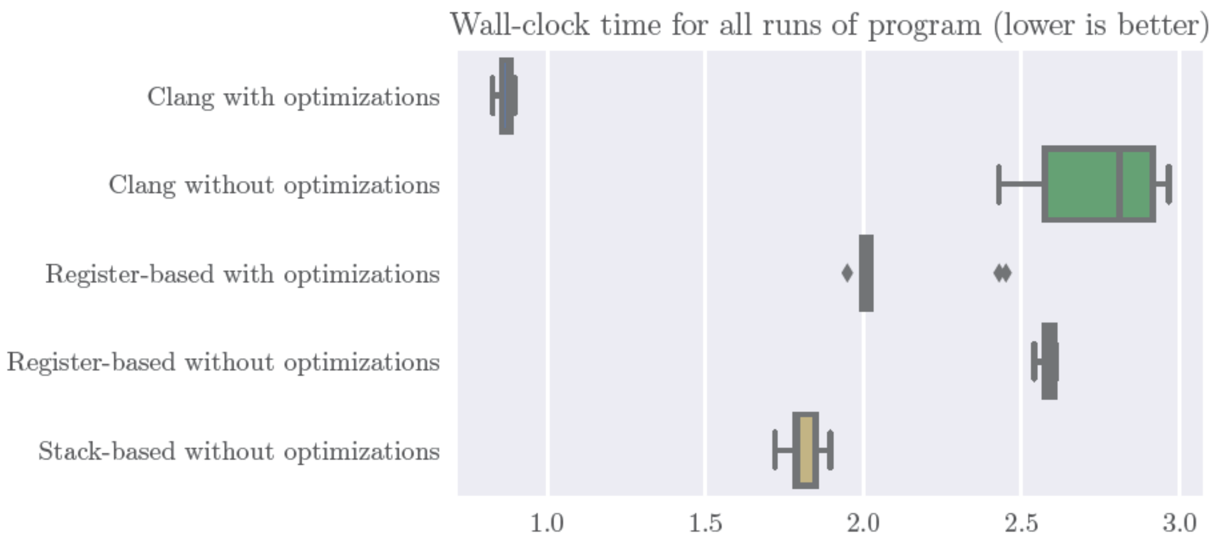
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	194	
Register-based without optimizations	126	35.05
Register-based with optimizations	126	0.



2.13 killerBubbles

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.809	5.489×10^{-2}
Register-based without optimizations	10	2.584	2.479×10^{-2}
Register-based with optimizations	10	2.088	1.868×10^{-1}
Clang without optimizations	10	2.747	1.941×10^{-1}
Clang with optimizations	10	8.693×10^{-1}	2.463×10^{-2}

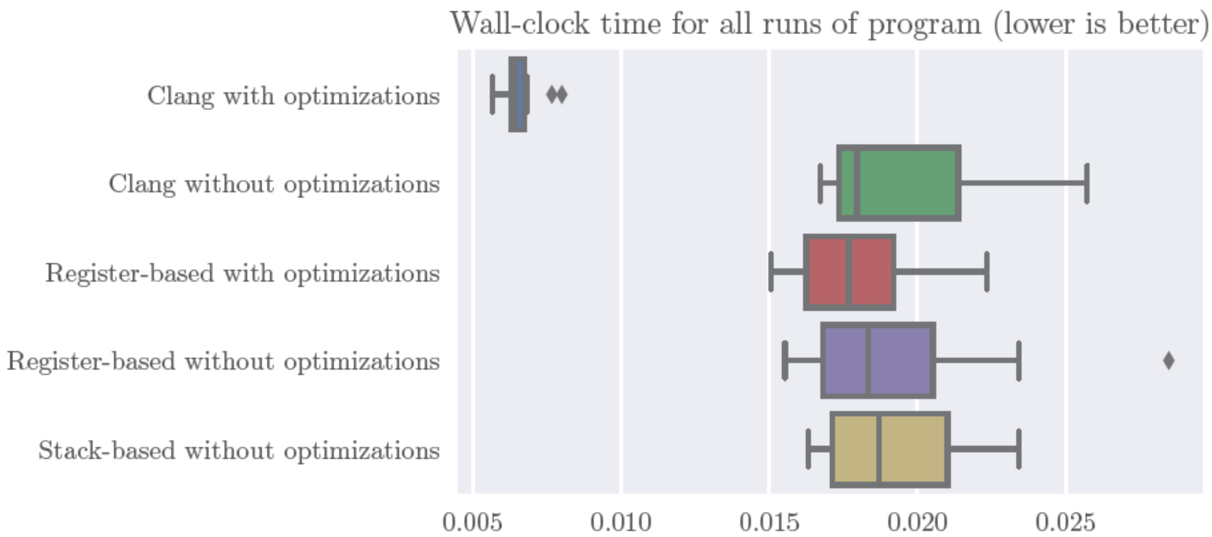
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	185	
Register-based without optimizations	96	48.11
Register-based with optimizations	93	3.125



2.14 mile1

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.925×10^{-2}	2.508×10^{-3}
Register-based without optimizations	10	1.944×10^{-2}	3.949×10^{-3}
Register-based with optimizations	10	1.811×10^{-2}	2.408×10^{-3}
Clang without optimizations	10	1.935×10^{-2}	3.076×10^{-3}
Clang with optimizations	10	6.601×10^{-3}	7.293×10^{-4}

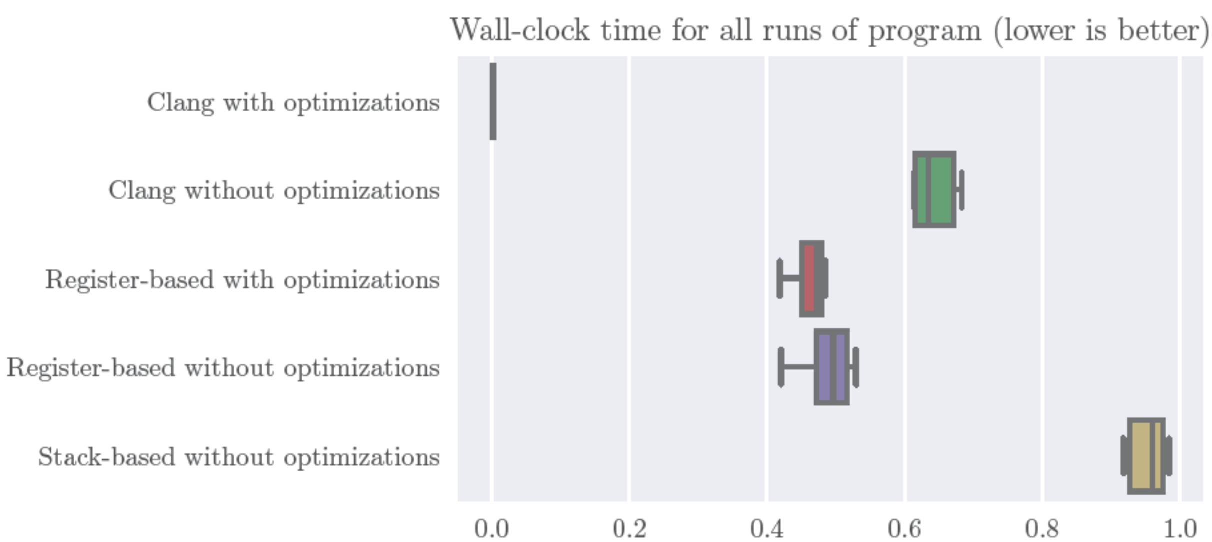
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	70	
Register-based without optimizations	33	52.86
Register-based with optimizations	31	6.061



2.15 mixed

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	9.518×10^{-1}	2.654×10^{-2}
Register-based without optimizations	10	4.903×10^{-1}	3.368×10^{-2}
Register-based with optimizations	10	4.607×10^{-1}	2.470×10^{-2}
Clang without optimizations	10	6.425×10^{-1}	2.972×10^{-2}
Clang with optimizations	10	2.013×10^{-3}	2.634×10^{-4}

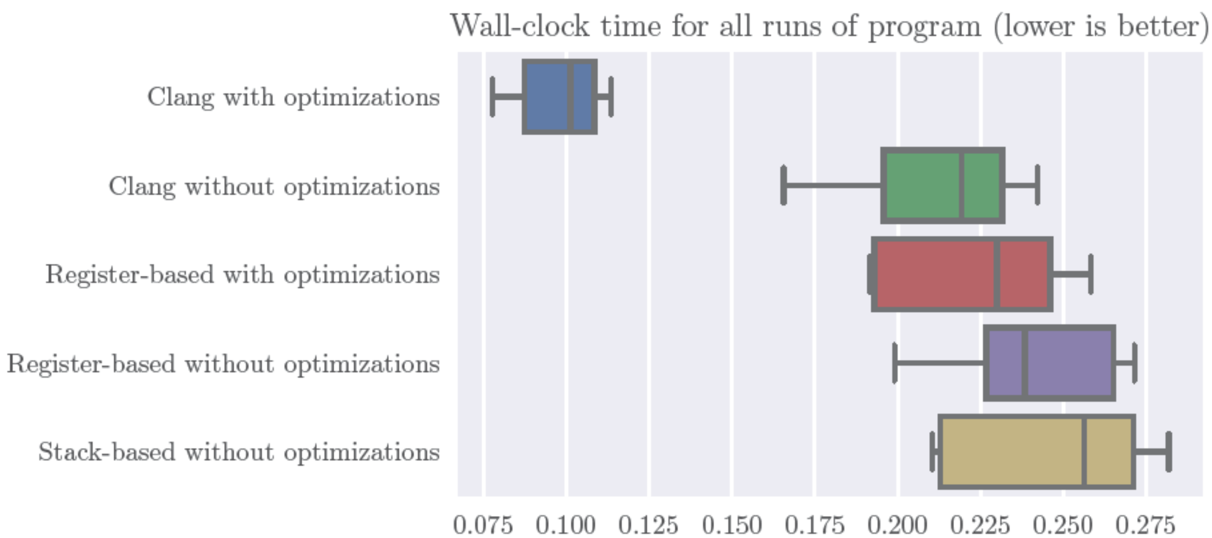
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	201	
Register-based without optimizations	116	42.29
Register-based with optimizations	96	17.24



2.16 primes

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	2.458×10^{-1}	3.041×10^{-2}
Register-based without optimizations	10	2.416×10^{-1}	2.442×10^{-2}
Register-based with optimizations	10	2.228×10^{-1}	2.762×10^{-2}
Clang without optimizations	10	2.127×10^{-1}	2.498×10^{-2}
Clang with optimizations	10	9.811×10^{-2}	1.301×10^{-2}

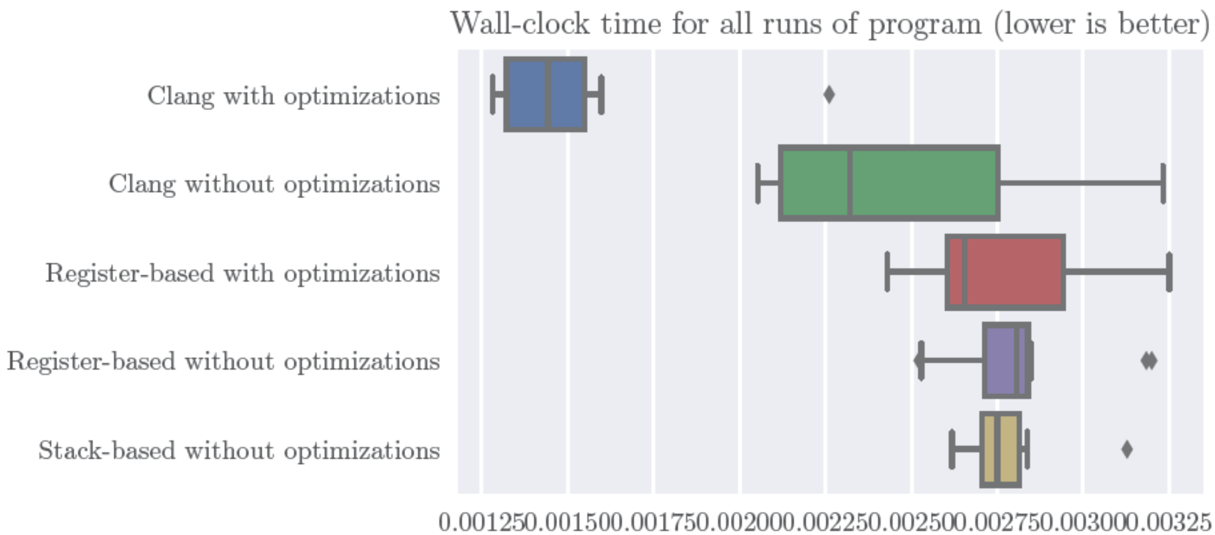
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	87	
Register-based without optimizations	38	56.32
Register-based with optimizations	38	0.



2.17 programBreaker

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	2.777×10^{-3}	1.405×10^{-4}
Register-based without optimizations	10	2.819×10^{-3}	2.282×10^{-4}
Register-based with optimizations	10	2.760×10^{-3}	2.806×10^{-4}
Clang without optimizations	10	2.447×10^{-3}	4.070×10^{-4}
Clang with optimizations	10	1.507×10^{-3}	2.898×10^{-4}

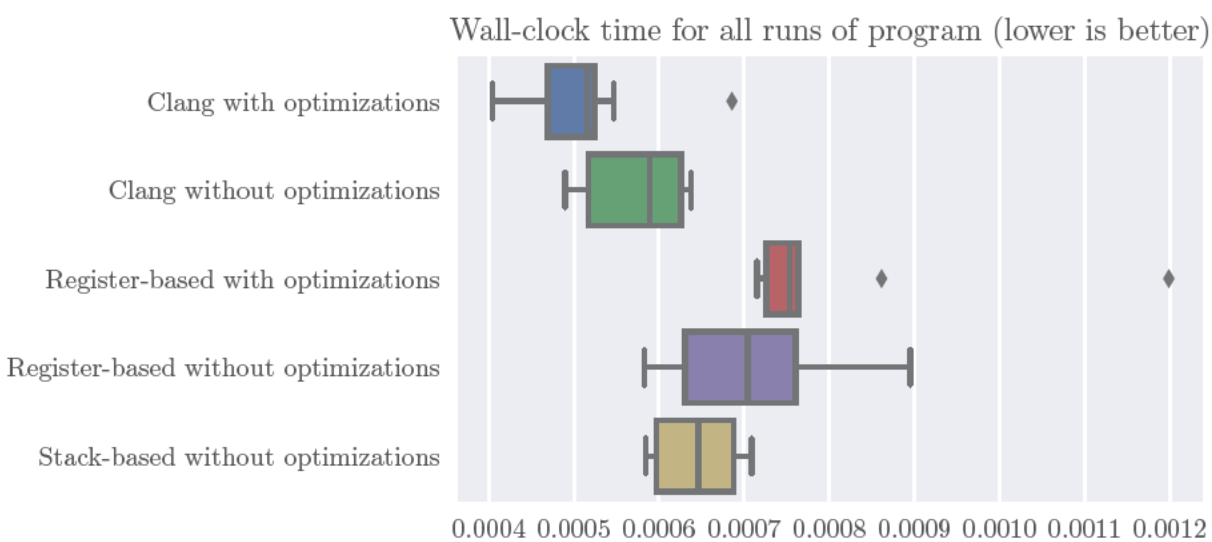
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	69	
Register-based without optimizations	33	52.17
Register-based with optimizations	31	6.061



2.18 stats

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	6.438×10^{-4}	4.921×10^{-5}
Register-based without optimizations	10	7.148×10^{-4}	1.017×10^{-4}
Register-based with optimizations	10	7.987×10^{-4}	1.468×10^{-4}
Clang without optimizations	10	5.725×10^{-4}	5.830×10^{-5}
Clang with optimizations	10	5.117×10^{-4}	7.439×10^{-5}

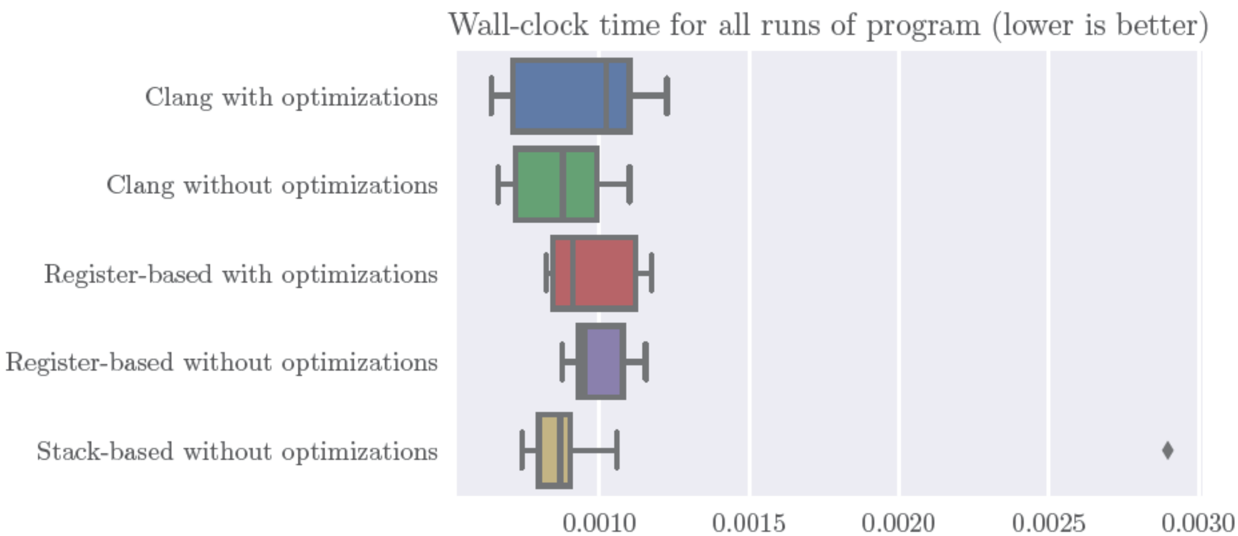
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	232	
Register-based without optimizations	101	56.47
Register-based with optimizations	101	0.



2.19 uncreativeBenchmark

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	1.059×10^{-3}	6.522×10^{-4}
Register-based without optimizations	10	9.949×10^{-4}	9.713×10^{-5}
Register-based with optimizations	10	9.727×10^{-4}	1.467×10^{-4}
Clang without optimizations	10	8.684×10^{-4}	1.588×10^{-4}
Clang with optimizations	10	9.309×10^{-4}	2.235×10^{-4}

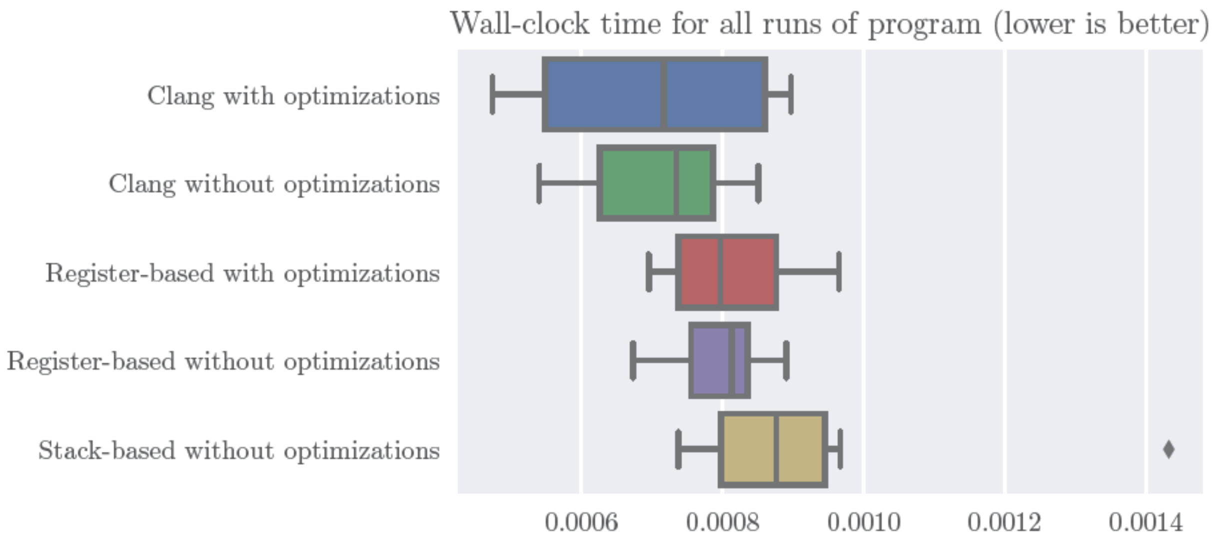
Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	298	
Register-based without optimizations	206	30.87
Register-based with optimizations	206	0.



2.20 wasteOfCycles

Type	Runs	Average Time (s)	Sample Std. Dev. (s)
Stack-based without optimizations	10	9.152×10^{-4}	1.993×10^{-4}
Register-based without optimizations	10	7.955×10^{-4}	6.660×10^{-5}
Register-based with optimizations	10	8.087×10^{-4}	9.029×10^{-5}
Clang without optimizations	10	7.115×10^{-4}	1.044×10^{-4}
Clang with optimizations	10	7.020×10^{-4}	1.742×10^{-4}

Type	Instruction Count	Percent improvement over previous
Stack-based without optimizations	42	
Register-based without optimizations	20	52.38
Register-based with optimizations	20	0.



References

[1] Matthias Braun et al. “Simple and Efficient Construction of Static Single Assignment Form.” In: (). Ed. by Ranjit Jhala and Koen De Bosschere. URL: http://dblp.uni-trier.de/db/conf/cc/cc2013.html#BraunBHLMZ13;%20http://dx.doi.org/10.1007/978-3-642-37051-9_6.

[2] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd ed. Morgan Kaufmann, Jan. 12, 2004, p. 800. ISBN: 1-55860-699-8.