

# Final Paper

CSC 431

Benjamin Kunkle

Dylan Liefer-Ives

Spencer Perley

Friday June 14th, 2024

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
1.1	Parsing . . . . .	2
1.2	Static Semantics: . . . . .	2
1.2.1	Type Checking . . . . .	2
1.2.2	Return path checking . . . . .	2
1.3	Intermediate Representation . . . . .	2
1.4	Optimizations: . . . . .	3
1.4.1	Sparse Conditional Constant Propagation . . . . .	3
1.4.2	Comparison Propagation . . . . .	3
1.4.3	Dead Code Elimination . . . . .	3
1.4.4	Empty Block Removal . . . . .	3
<b>2</b>	<b>Analysis</b>	<b>4</b>
2.1	BenchMarkishTopics . . . . .	4
2.2	Fibonacci . . . . .	5
2.3	GeneralFunctAndOptimize . . . . .	6
2.4	OptimizationBenchmark . . . . .	7
2.5	TicTac . . . . .	8
2.6	array_sort . . . . .	9
2.7	array_sum . . . . .	10
2.8	bert . . . . .	11
2.9	biggest . . . . .	12
2.10	binaryConverter . . . . .	13
2.11	brett . . . . .	14
2.12	creativeBenchMarkName . . . . .	15
2.13	fact_sum . . . . .	16
2.14	hailstone . . . . .	17
2.15	hanoi_benchmark . . . . .	18
2.16	killerBubbles . . . . .	19
2.17	mile1 . . . . .	20
2.18	mixed . . . . .	21
2.19	primes . . . . .	22
2.20	programBreaker . . . . .	23
2.21	stats . . . . .	24
2.22	wasteOfCycles . . . . .	25

# 1 Overview

## 1.1 Parsing

For our compiler, we decided not to use ANTLR or any other parser generator. Instead, we wrote our own tokenizer and parser.

```
1 pub const Node = struct {  
2     kind: Kind,  
3     token: Token,  
4 }
```

Figure 1: INSERT FIGURE DESCRIPTION HERE

```
1 pub const Parser = struct {  
2     tokens: []Token,  
3     input: []const u8,  
4  
5     ast: NodeList,  
6     astLen: usize = 0,  
7  
8     pos: usize = 0,  
9     readPos: usize = 1,  
10  
11     allocator: std.mem.Allocator,  
12 }
```

Figure 2: INSERT FIGURE DESCRIPTION HERE

```
1 pub const Range = struct {  
2     start: u32,  
3     end: u32,  
4 }
```

Figure 3: INSERT FIGURE DESCRIPTION HERE

## 1.2 Static Semantics:

### 1.2.1 Type Checking

### 1.2.2 Return path checking

## 1.3 Intermediate Representation

Our compiler stores the IR as a list of functions. Each function then stores a list of all the registers, instructions, and basic blocks. Each basic block stores an ordered list of references to instructions. Each instruction stores references to the registers it uses as well as the register it writes to and any other instruction specific data.

```

1 pub const Token = struct {
2     kind: TokenKind,
3     _range: Range,
4 };

```

Figure 4: INSERT FIGURE DESCRIPTION HERE

```

1 While: struct {
2     /// The condition expression to check
3     cond: Ref(.Expression),
4     /// The block of code to execute
5     block: Ref(.Block),
6 },

```

Figure 5: INSERT FIGURE DESCRIPTION HERE

The IR is used as a control flow graph as well through the references which the basic blocks hold. This is especially useful for

## 1.4 Optimizations:

For our compiler, we implemented the following optimizations:

### 1.4.1 Sparse Conditional Constant Propagation

First, we implemented SCCP which propagates constant values. If while propagating a constant value, a conditional branch is reached, and the condition is known, we only continue propagating down that branch. Once we are finished any unreachable basic blocks are removed, and references to things in the blocks like phi nodes are replaced with the constant value.

### 1.4.2 Comparison Propagation

On top of SCCP, we implemented comparison propagation. This is an extension of SCCP that evaluates comparisons and removes redundant ones. For example, if we check that a value is less than some number and then check again if it is bigger than some larger number, we can remove the child comparison as it is impossible to be true. We also can create constants from direct comparisons. For example, if we compare a value to 0, we can do constant propagation on the value in the branch where it is the case where the value is 0.

### 1.4.3 Dead Code Elimination

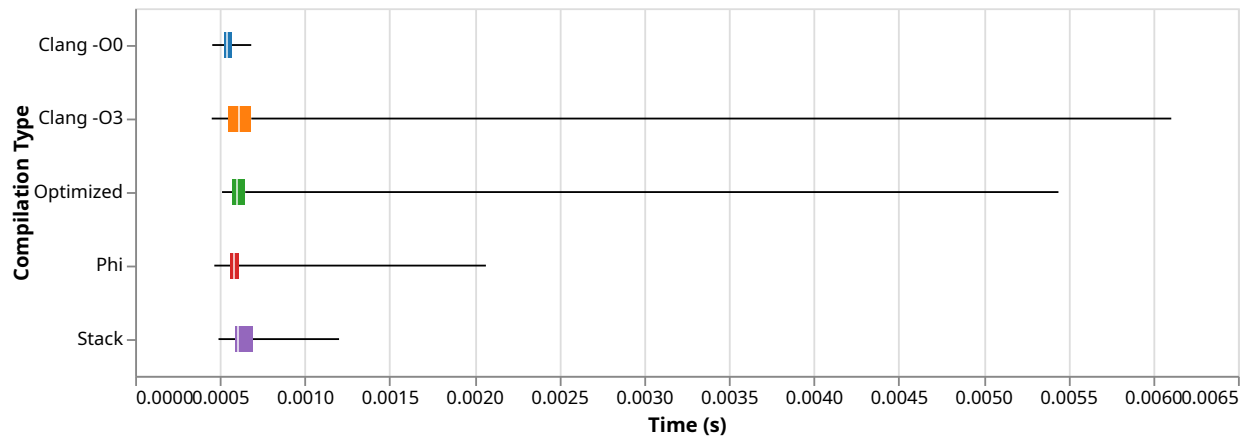
Finally, we implemented mark and sweep dead code elimination. This was fairly straightforward to implement. For each function in the IR, we first marked all the side effects like calls as well as the return value. Then we marked everything that the marked values relied on. Finally we removed anything not marked.

### 1.4.4 Empty Block Removal

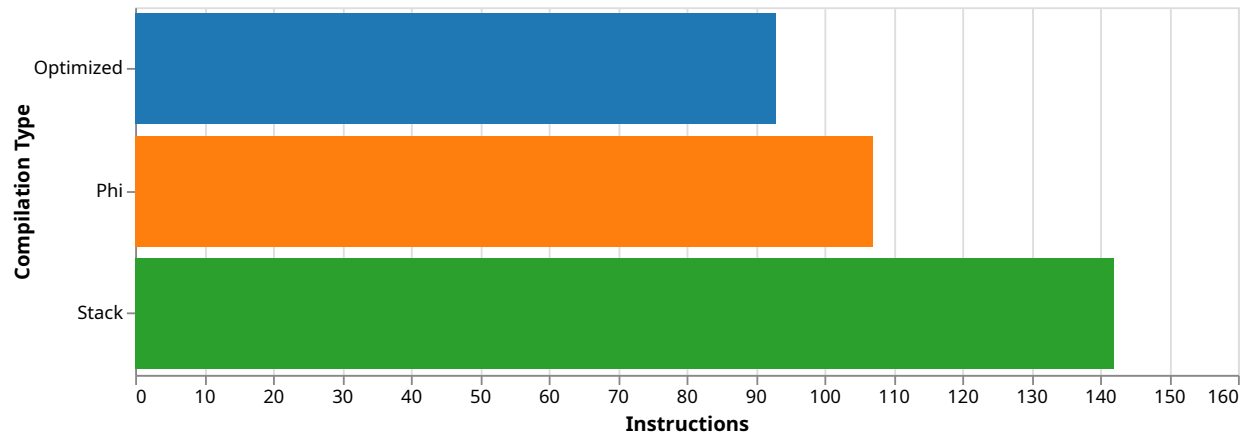
Somewhat related to dead code elimination, we also implemented empty block removal which removes any basic blocks with only a single jump instruction.

## 2 Analysis

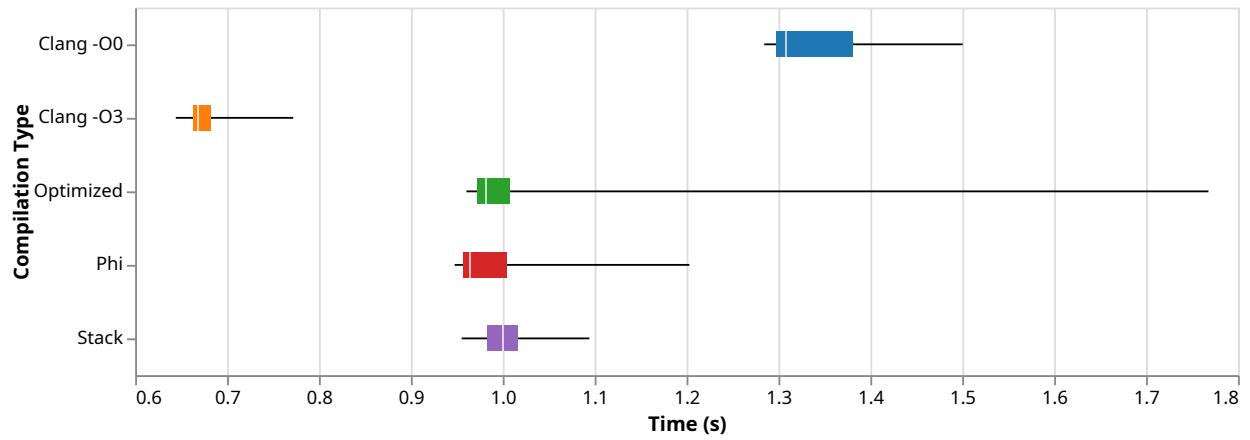
### 2.1 BenchMarkishTopics



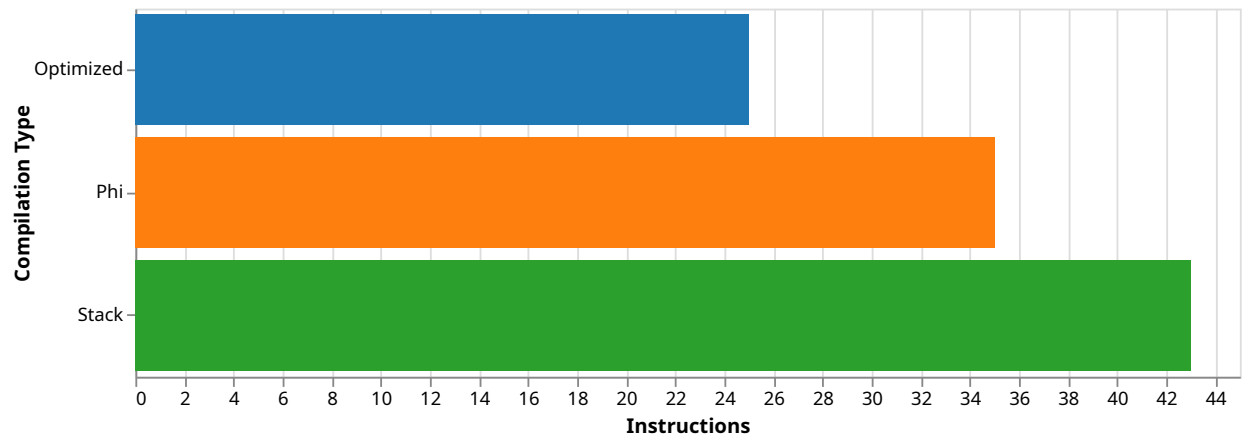
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.000548957	4.20744e-05	100
1	Clang -O3	0.000875332	0.000904613	100
2	Optimized	0.000765208	0.000645905	100
3	Phi	0.000602088	0.000154801	100
4	Stack	0.000656832	0.000130187	100



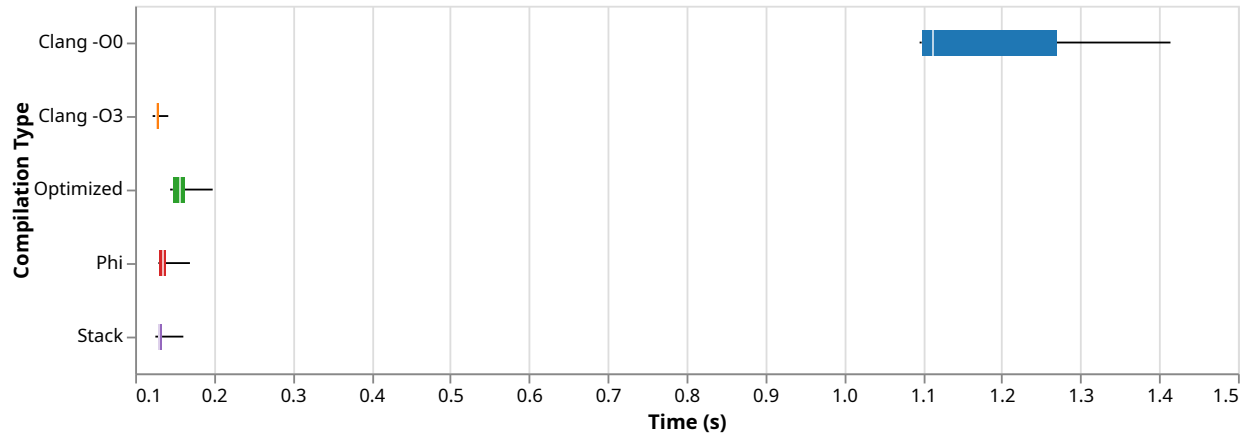
## 2.2 Fibonacci



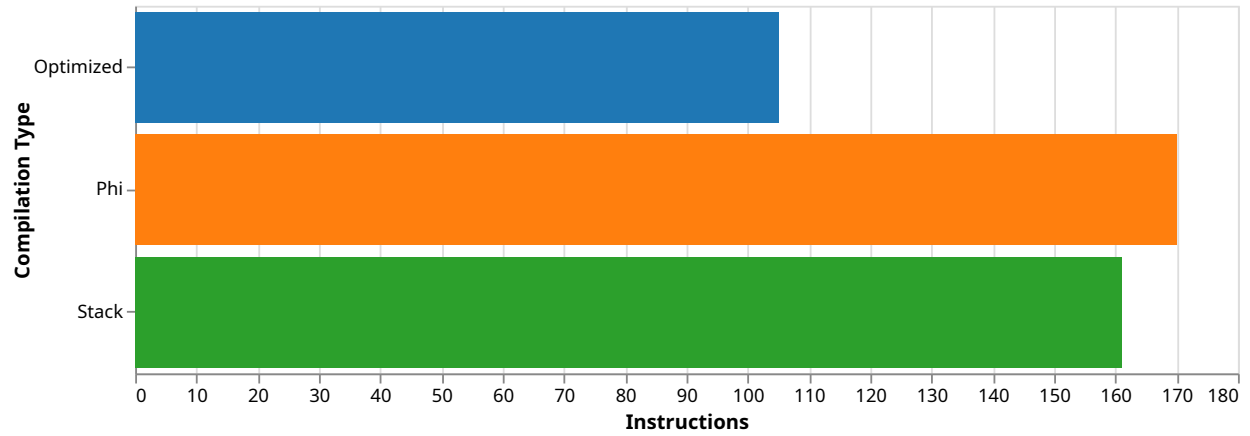
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	1.34265	0.06914	10
1	Clang -O3	0.686857	0.0448232	10
2	Optimized	1.1175	0.294606	10
3	Phi	0.999313	0.0796519	10
4	Stack	1.00541	0.0412773	10



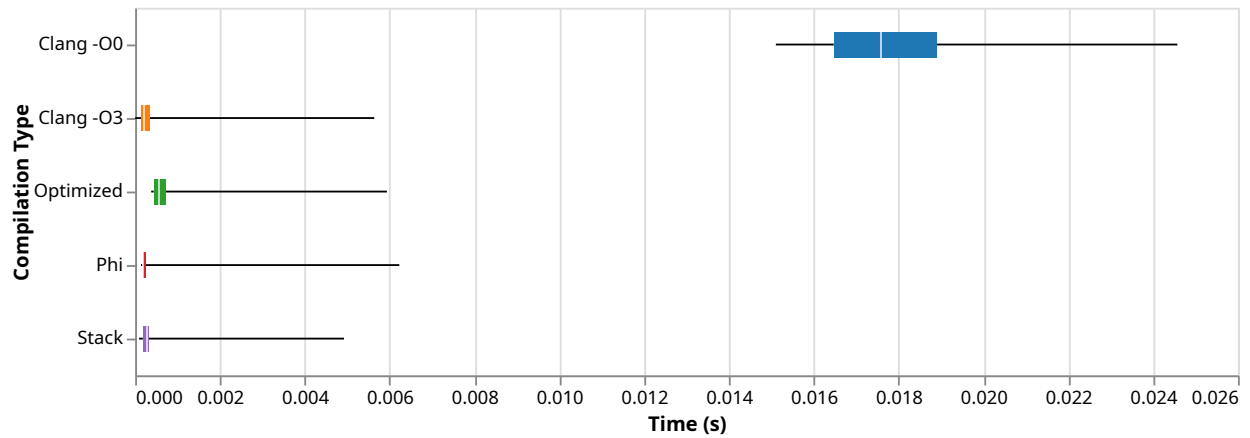
## 2.3 GeneralFuncAndOptimize



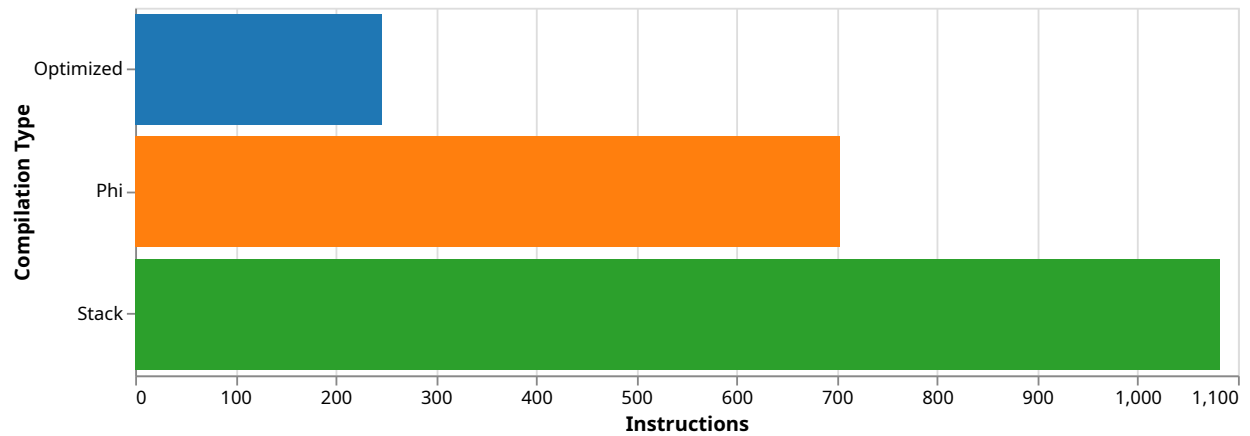
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	1.1972	0.131997	10
1	Clang -O3	0.128315	0.00548708	10
2	Optimized	0.160155	0.0166928	10
3	Phi	0.138412	0.0121598	10
4	Stack	0.13372	0.0100984	10



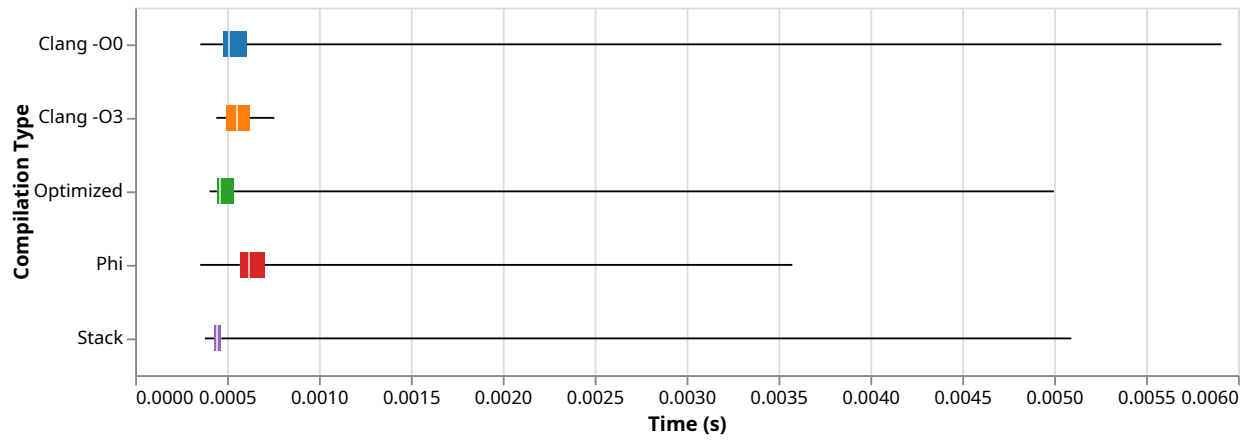
## 2.4 OptimizationBenchmark



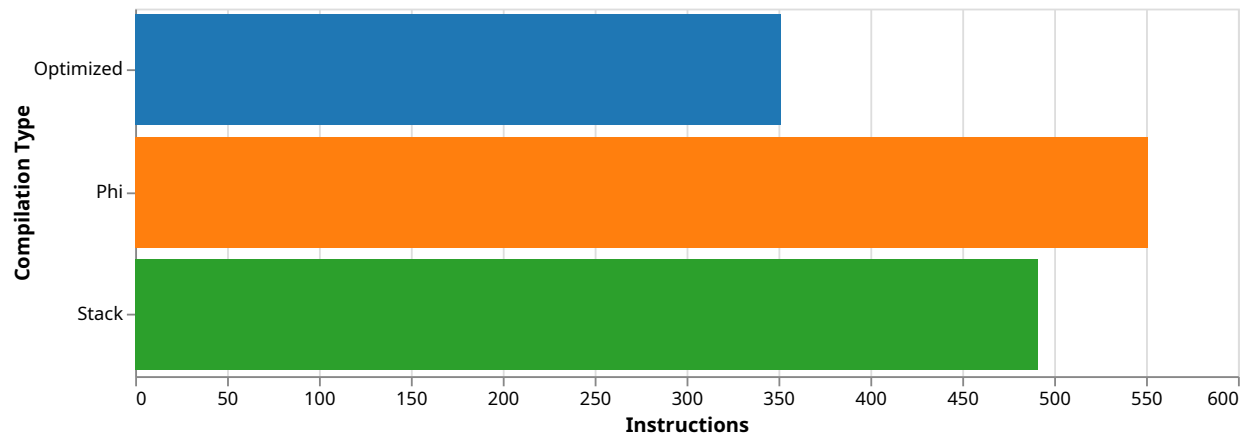
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.0180274	0.00196866	100
1	Clang -O3	0.000387386	0.000790512	100
2	Optimized	0.000707812	0.000695657	100
3	Phi	0.000664428	0.00130693	100
4	Stack	0.000310872	0.0004841	100



## 2.5 TicTac

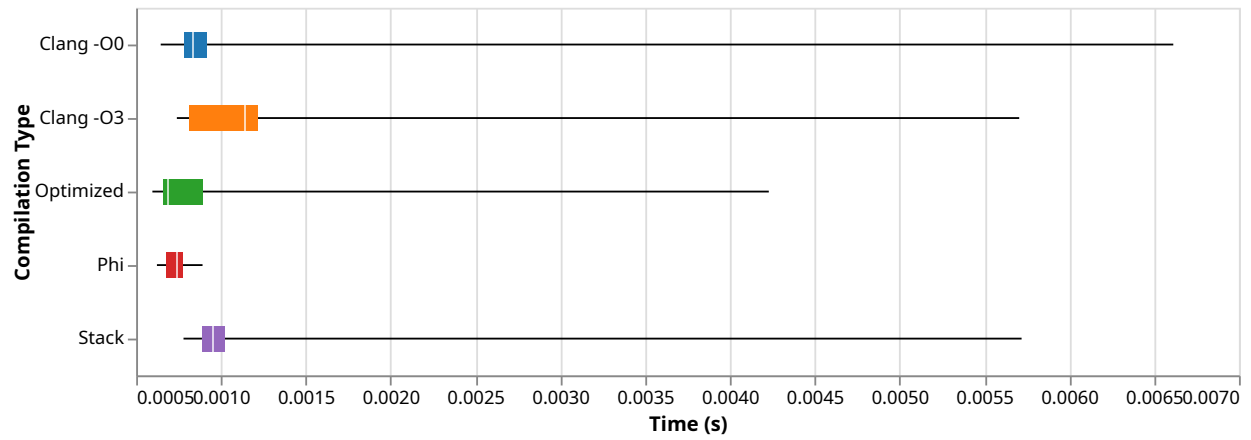


	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.000679915	0.000679702	100
1	Clang -O3	0.000568203	8.50207e-05	100
2	Optimized	0.000553835	0.000502394	100
3	Phi	0.00077026	0.000537269	100
4	Stack	0.000506808	0.000473132	100

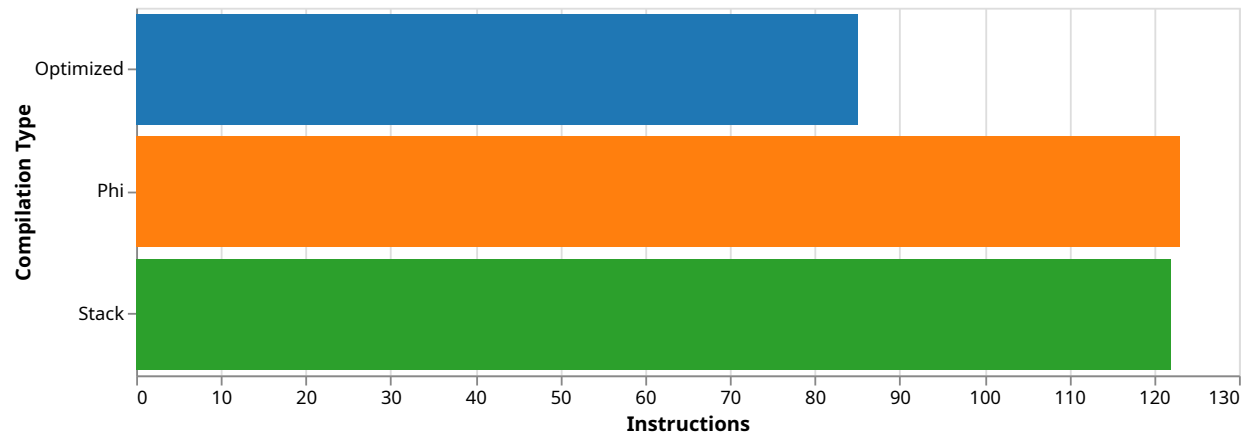




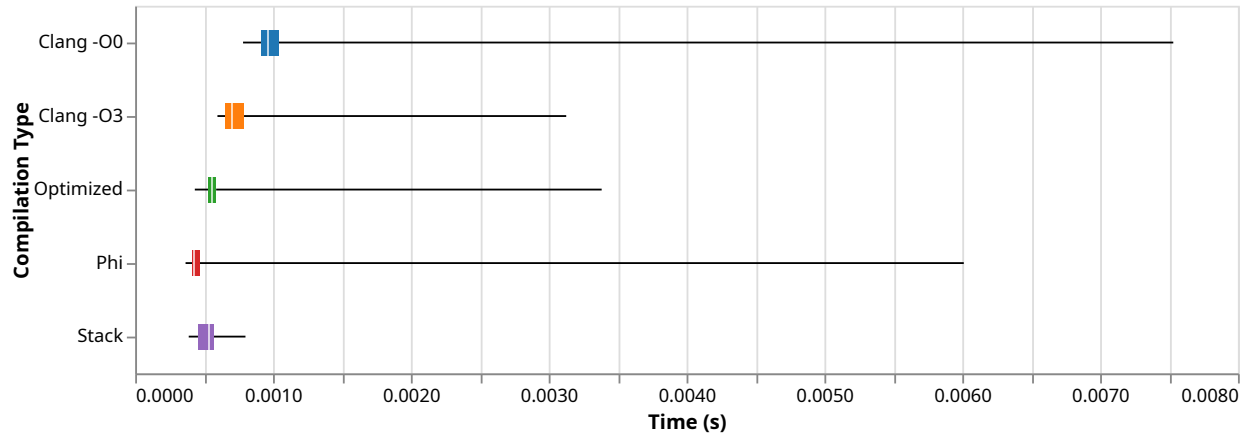
## 2.6 array\_sort



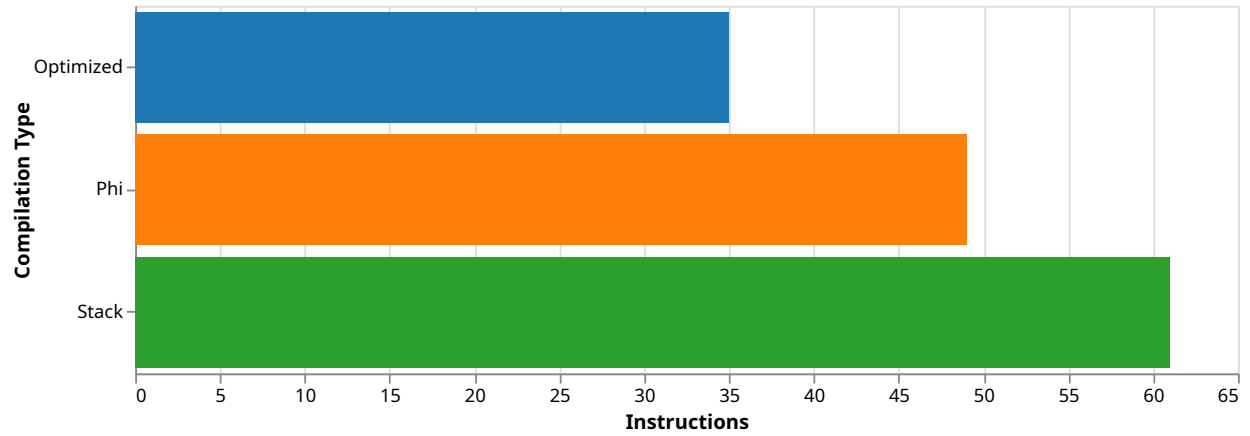
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00102434	0.000777098	100
1	Clang -O3	0.00155305	0.00124491	100
2	Optimized	0.000830775	0.000435514	100
3	Phi	0.000728558	6.07833e-05	100
4	Stack	0.00108512	0.000680252	100



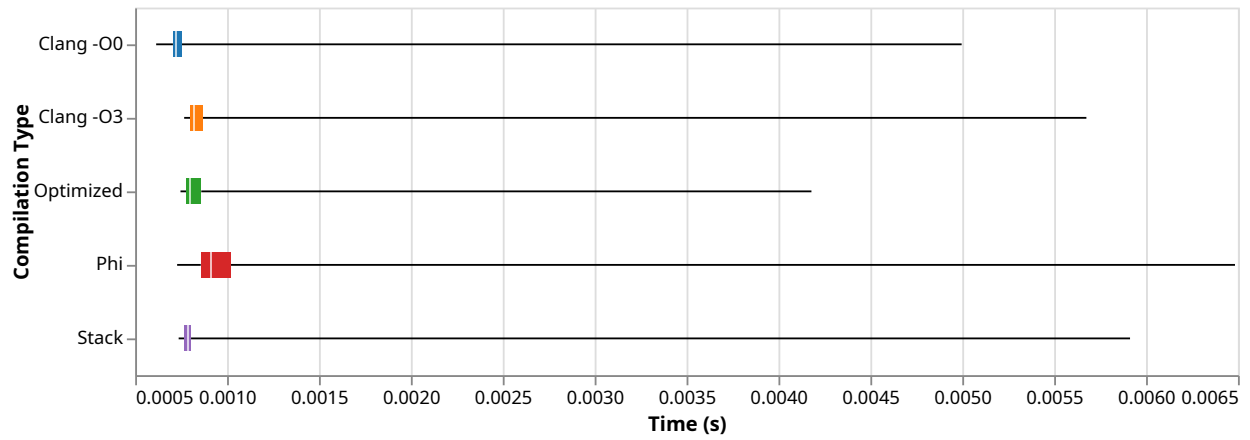
## 2.7 array\_sum



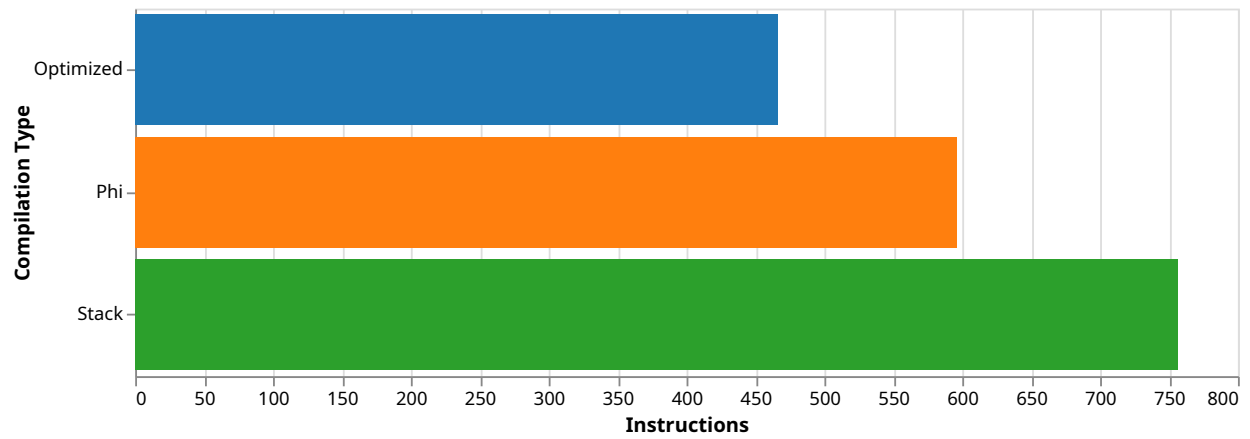
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00108382	0.000731937	100
1	Clang -O3	0.000811957	0.000413607	100
2	Optimized	0.000627221	0.000388521	100
3	Phi	0.000600105	0.000721709	100
4	Stack	0.000529111	8.06176e-05	100



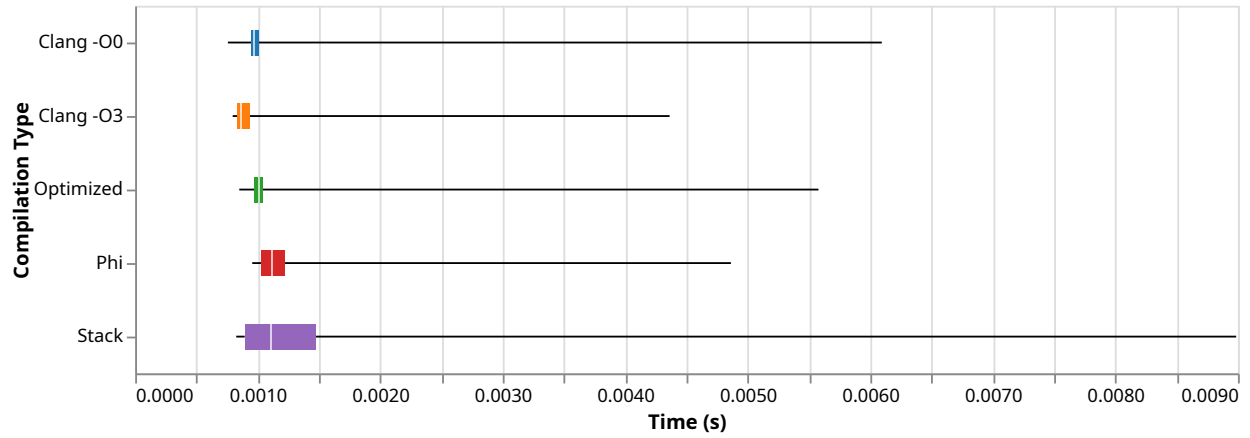
## 2.8 bert



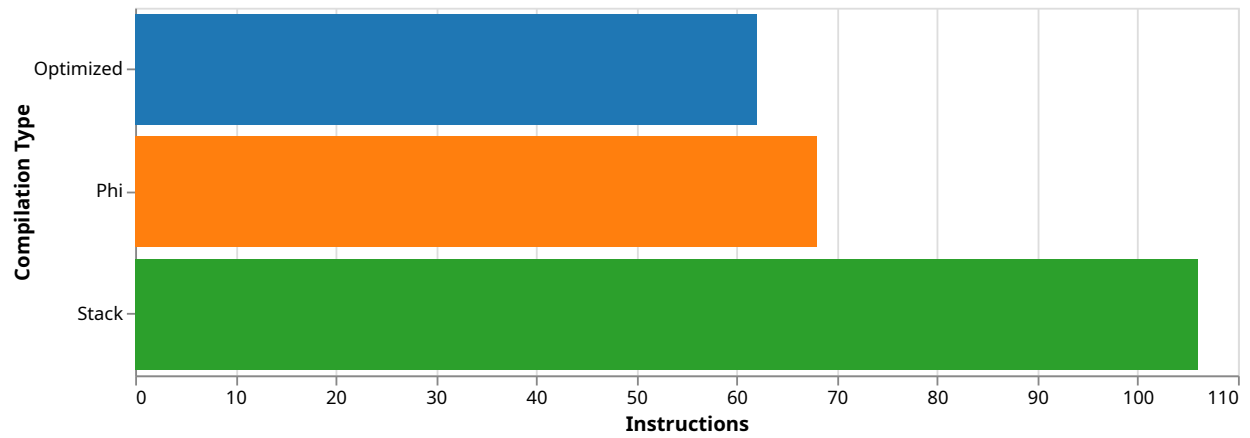
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.000842613	0.000569818	100
1	Clang -O3	0.000937838	0.000563945	100
2	Optimized	0.000871462	0.000390434	100
3	Phi	0.001087	0.000820705	100
4	Stack	0.000848477	0.000515771	100



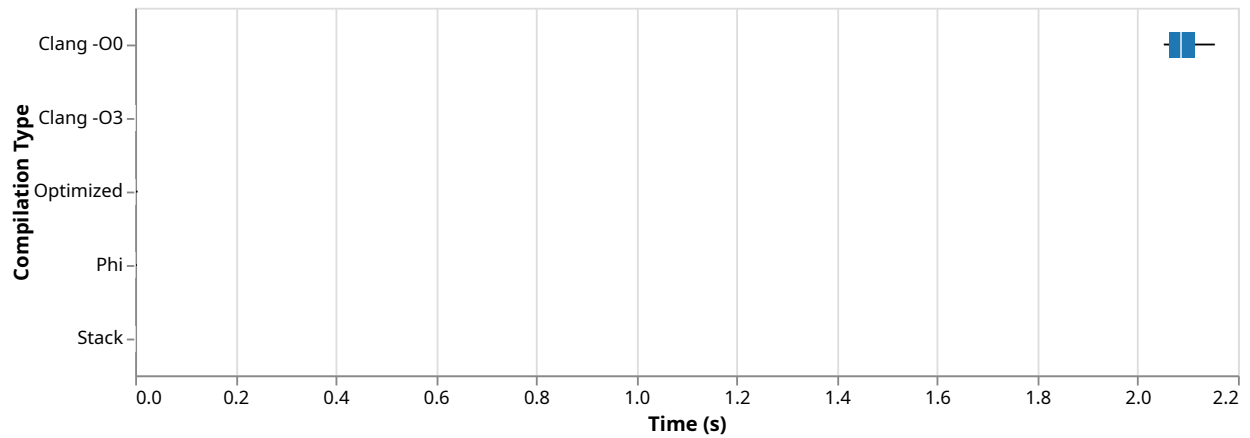
## 2.9 biggest



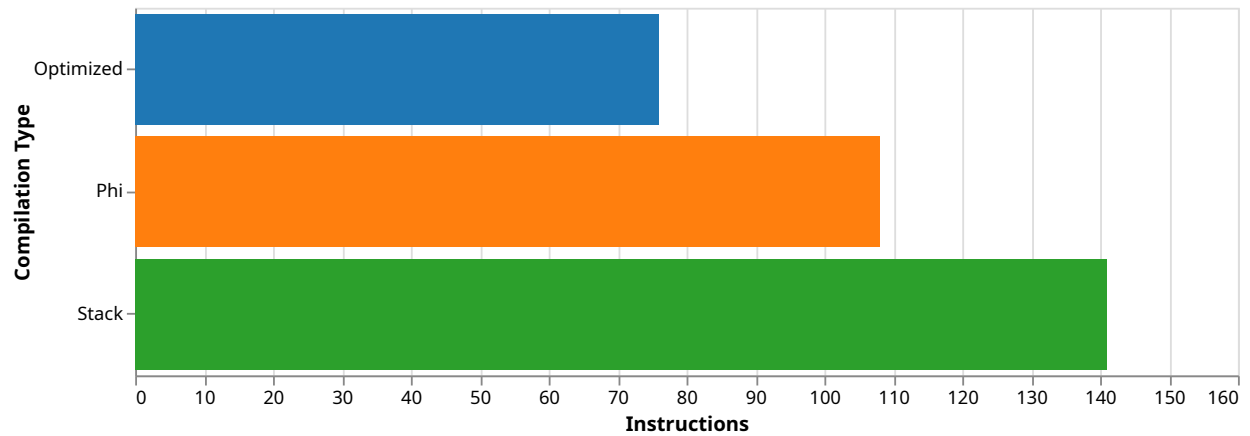
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00113256	0.000766537	100
1	Clang -O3	0.000949541	0.000442268	100
2	Optimized	0.00113283	0.00059959	100
3	Phi	0.00116739	0.000387661	100
4	Stack	0.00147517	0.00126153	100



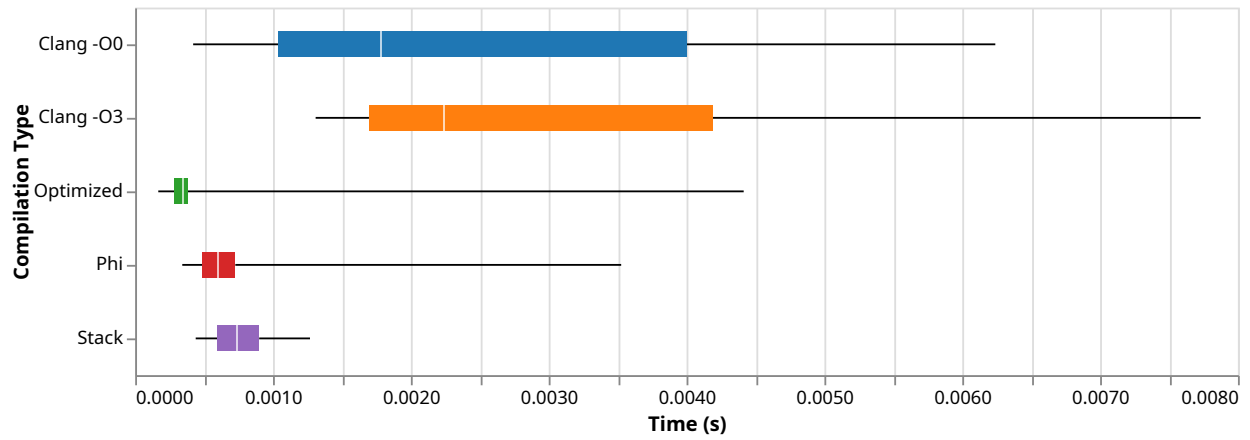
## 2.10 binaryConverter



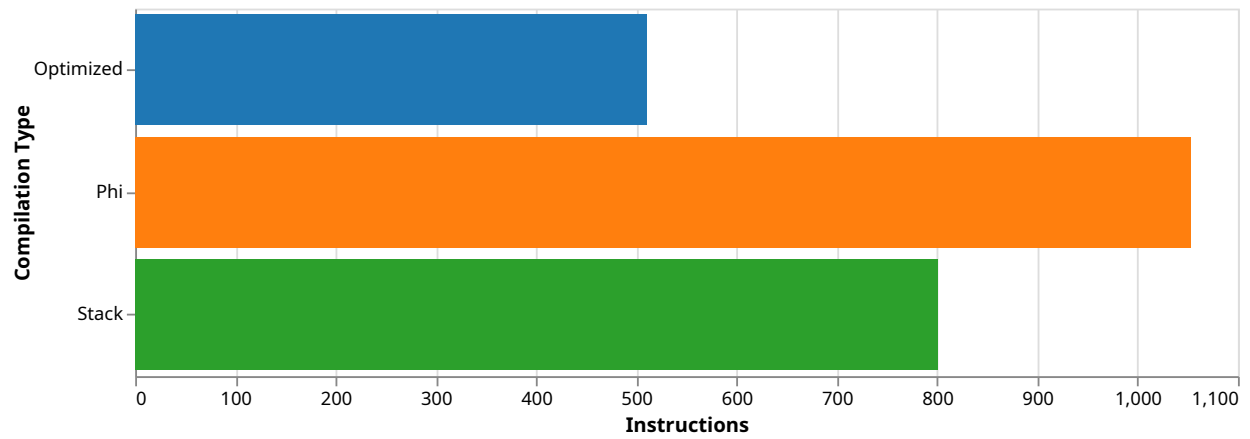
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	2.09172	0.0362483	10
1	Clang -O3	0.000170964	1.44655e-05	10
2	Optimized	0.00119744	0.00149786	10
3	Phi	0.000996188	0.00119554	10
4	Stack	0.000306354	4.05001e-05	10



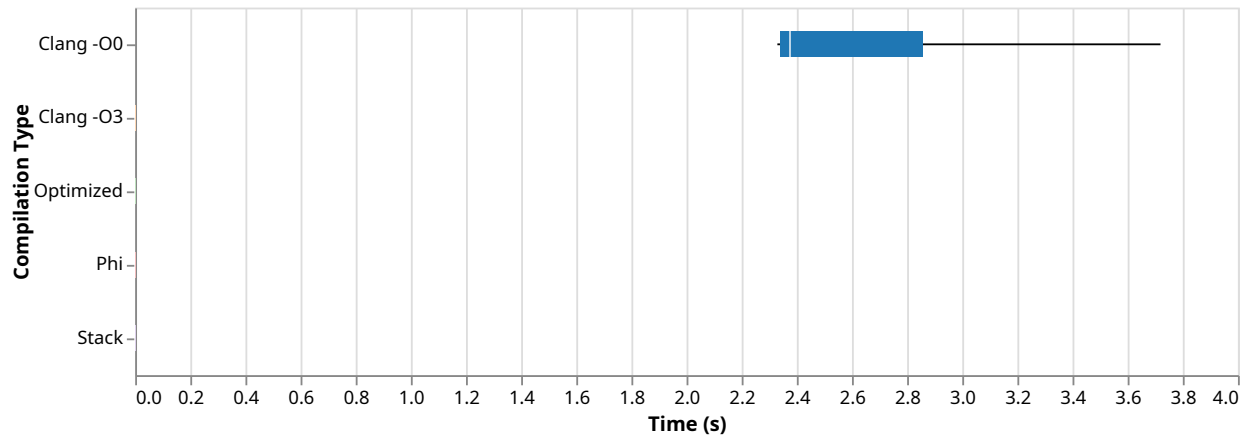
## 2.11 brett



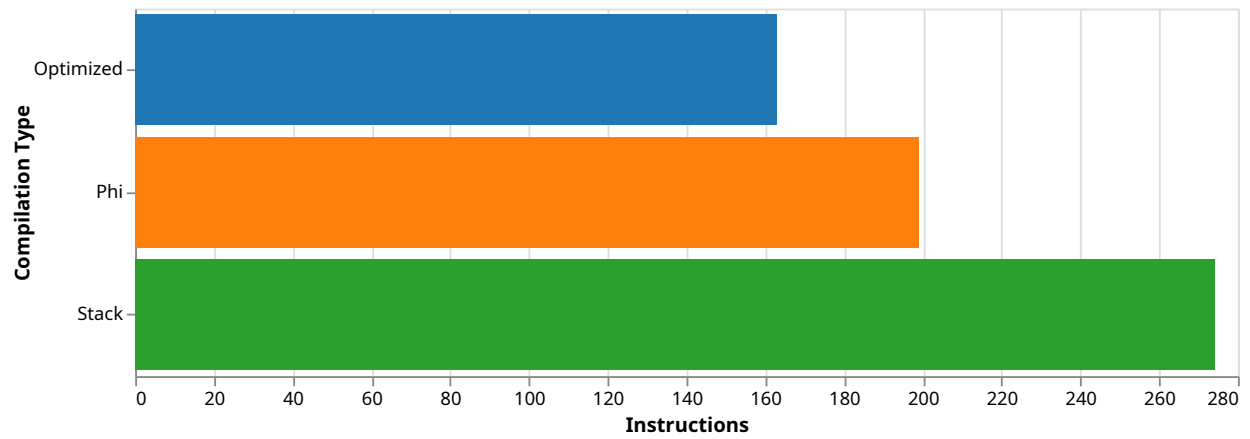
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00256278	0.0018535	100
1	Clang -O3	0.00311457	0.00203451	100
2	Optimized	0.000392386	0.000441803	100
3	Phi	0.000662178	0.000405433	100
4	Stack	0.000771347	0.000215964	100



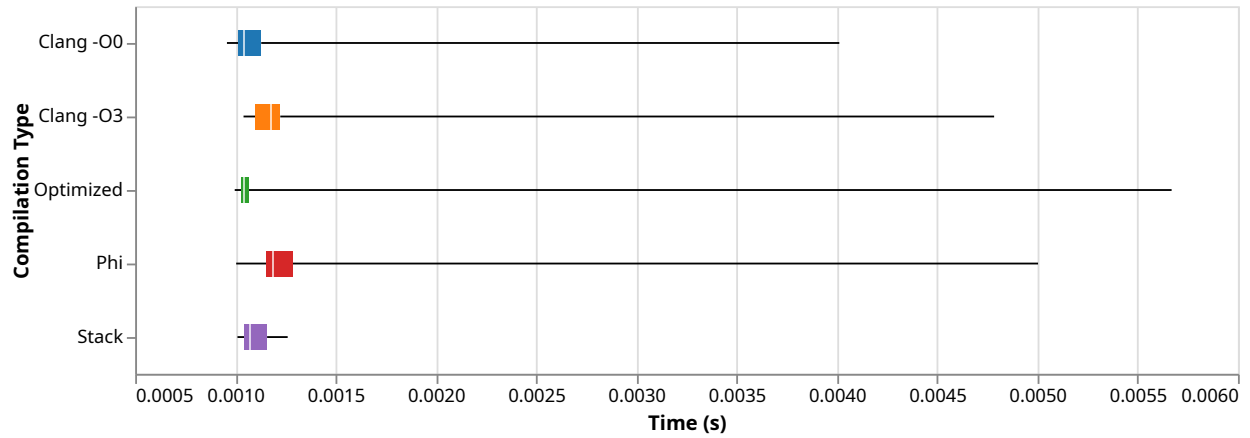
## 2.12 creativeBenchMarkName



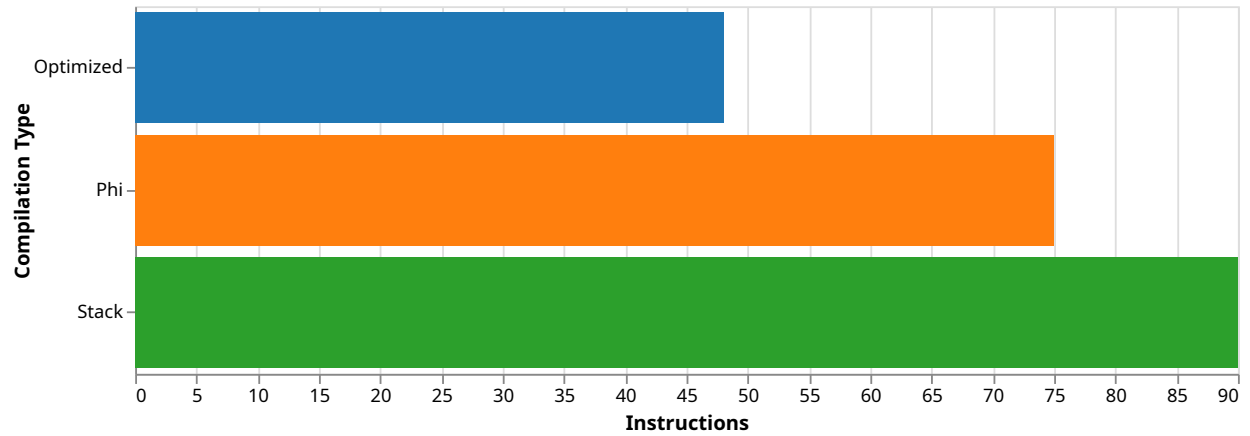
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	2.67323	0.53446	10
1	Clang -O3	0.000504582	1.66441e-05	10
2	Optimized	0.000570717	3.12939e-05	10
3	Phi	0.000562817	3.55185e-05	10
4	Stack	0.000579521	3.73848e-05	10



## 2.13 fact\_sum

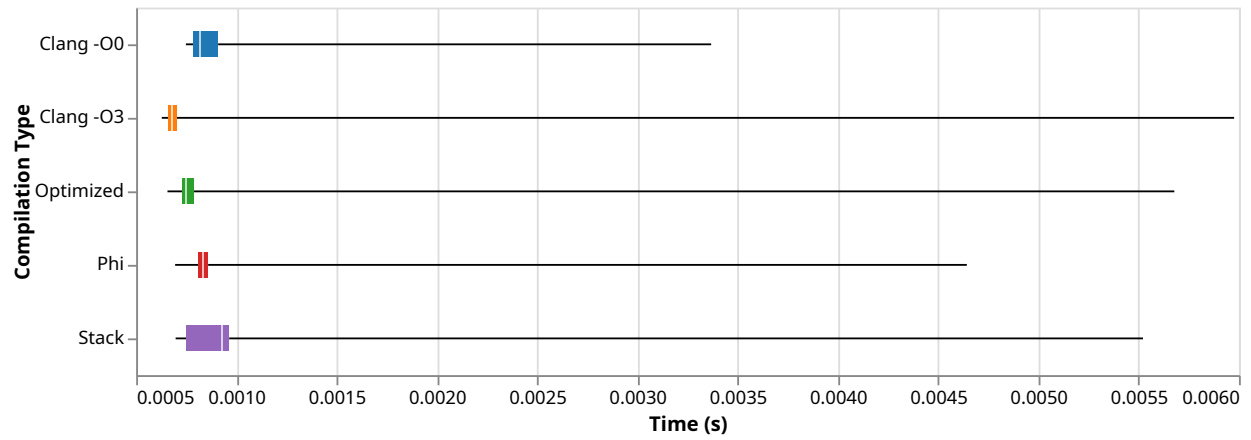


	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.0012927	0.000623988	100
1	Clang -O3	0.00122239	0.000407783	100
2	Optimized	0.00110627	0.000471794	100
3	Phi	0.00137334	0.000660169	100
4	Stack	0.00109687	6.46681e-05	100

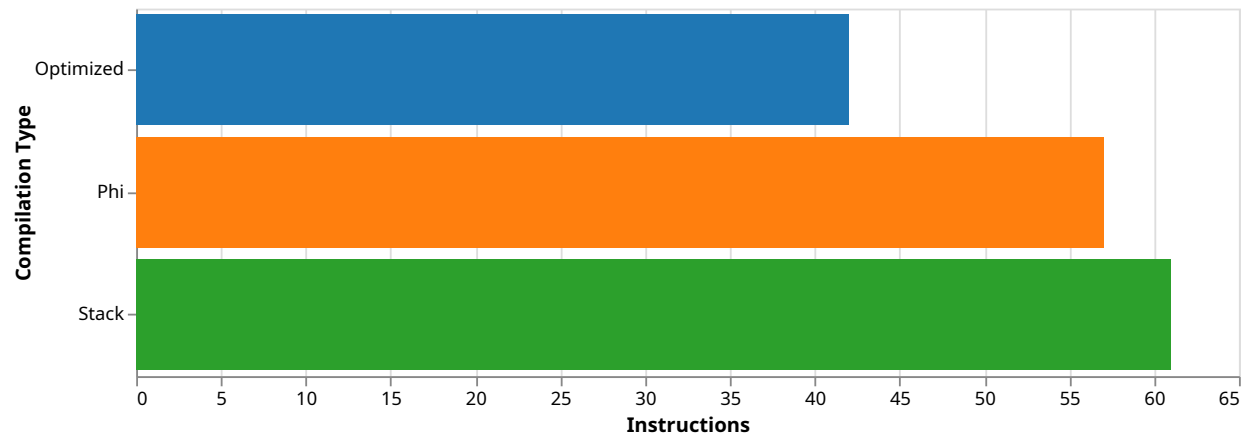




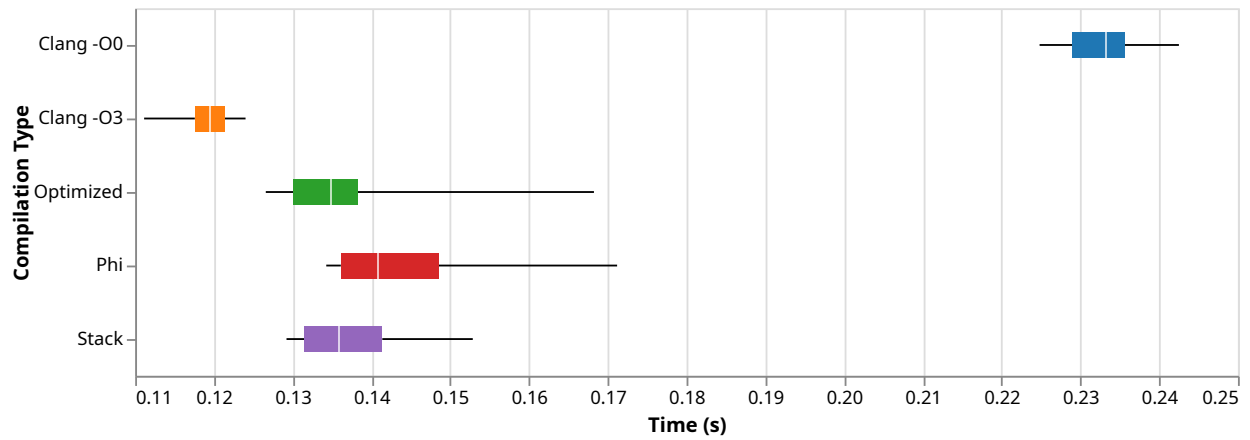
## 2.14 hailstone



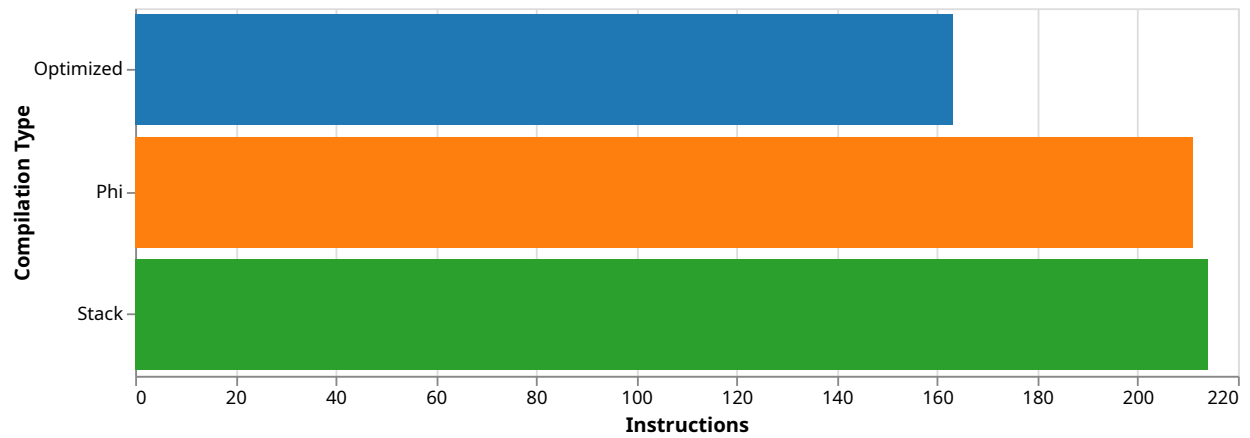
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.000902551	0.000359201	100
1	Clang -O3	0.000765127	0.000565975	100
2	Optimized	0.000907937	0.000724356	100
3	Phi	0.000879652	0.000405196	100
4	Stack	0.0010427	0.000755512	100



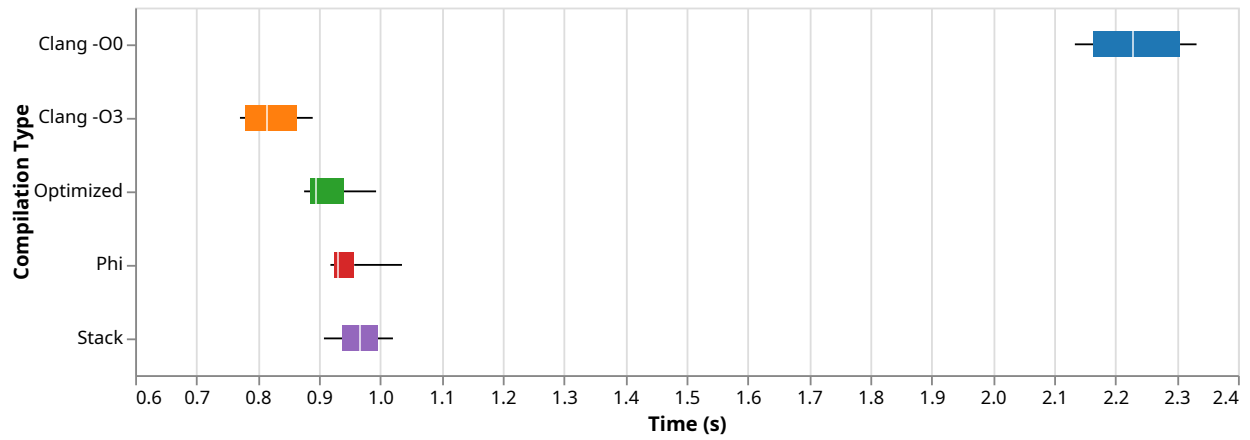
## 2.15 hanoi\_benchmark



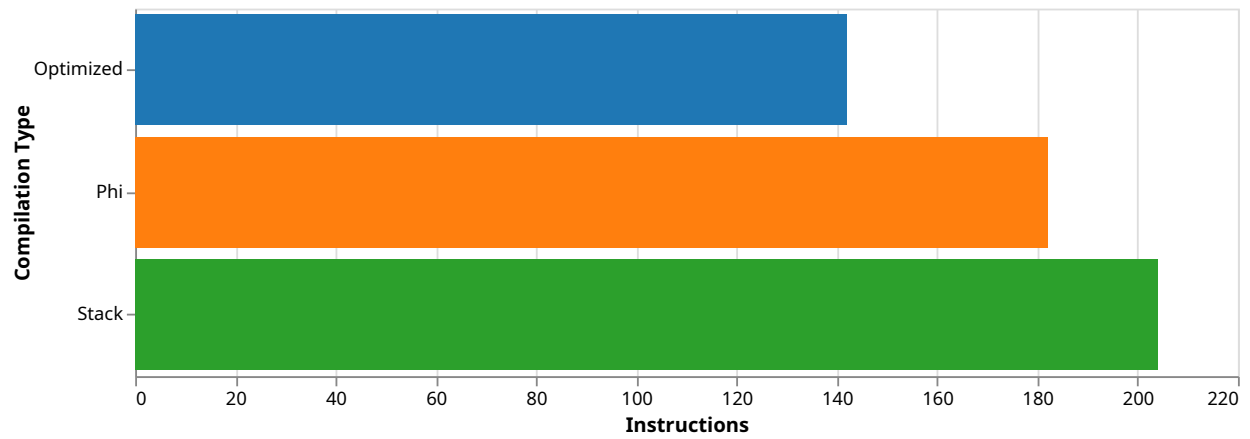
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.232941	0.00549127	10
1	Clang -O3	0.118951	0.00363262	10
2	Optimized	0.137448	0.0121549	10
3	Phi	0.144565	0.0118456	10
4	Stack	0.137341	0.00766898	10



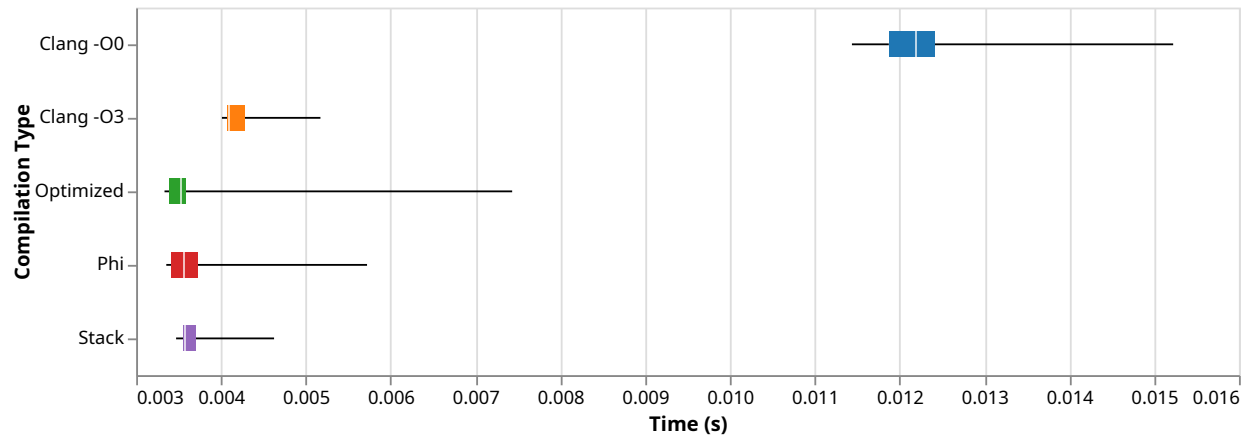
## 2.16 killerBubbles



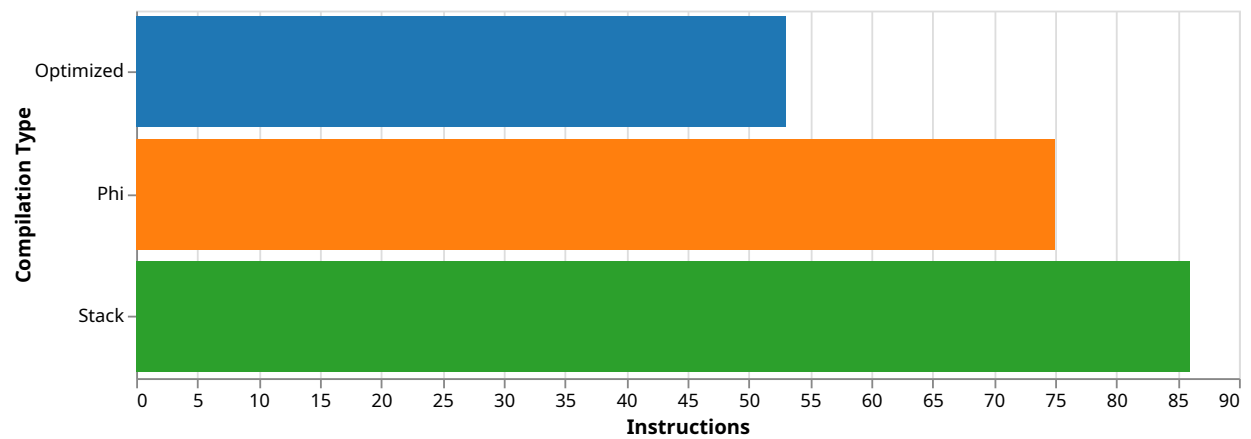
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	2.23463	0.0805044	10
1	Clang -O3	0.822052	0.0471532	10
2	Optimized	0.912434	0.0407113	10
3	Phi	0.947376	0.0360129	10
4	Stack	0.966933	0.0396408	10



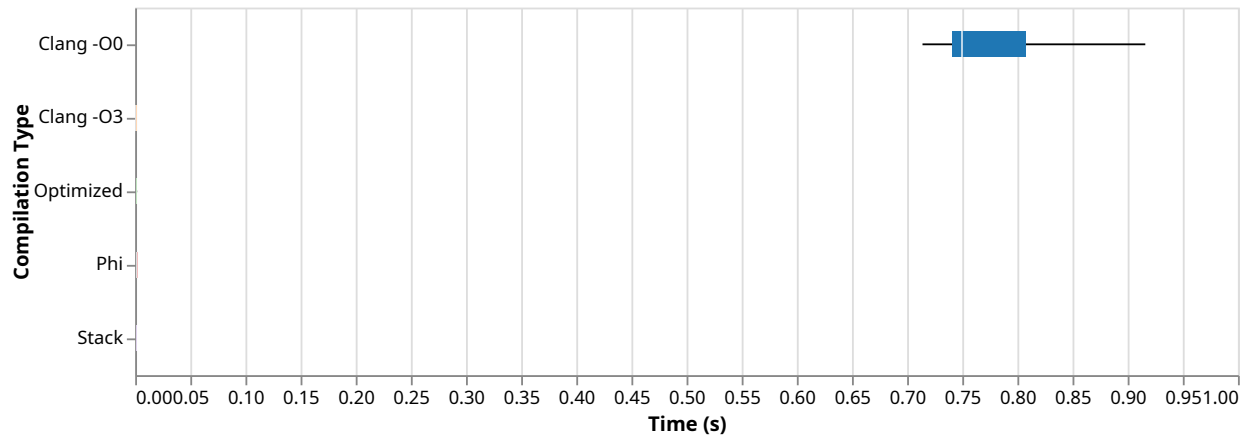
## 2.17 mile1



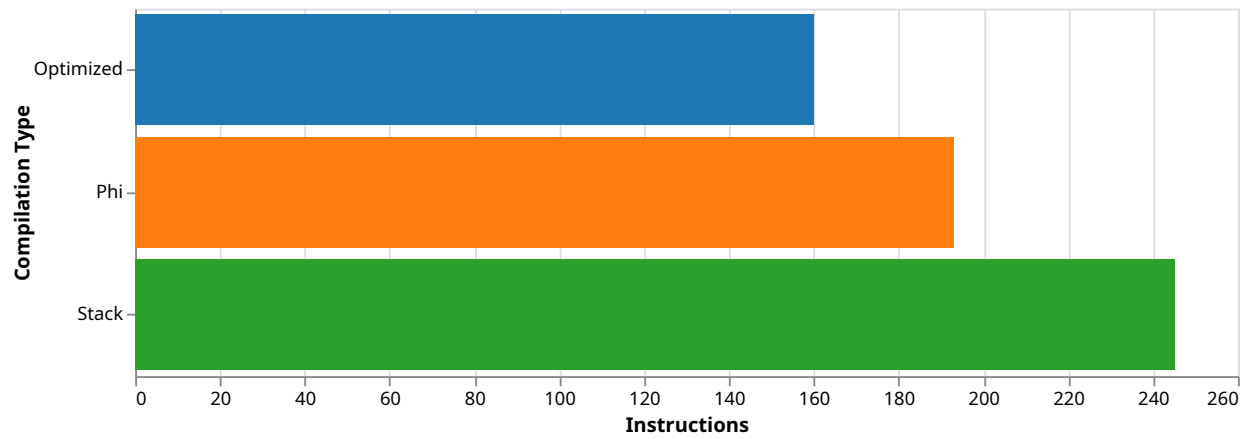
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.0122475	0.000701151	100
1	Clang -O3	0.00418146	0.000174946	100
2	Optimized	0.00359542	0.000495696	100
3	Phi	0.00359835	0.000275374	100
4	Stack	0.00363771	0.000162381	100



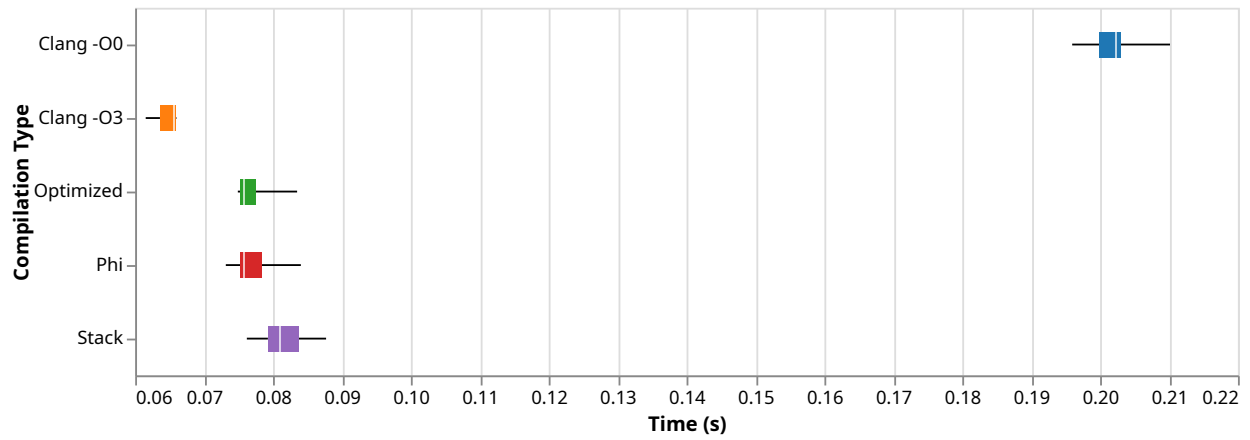
## 2.18 mixed



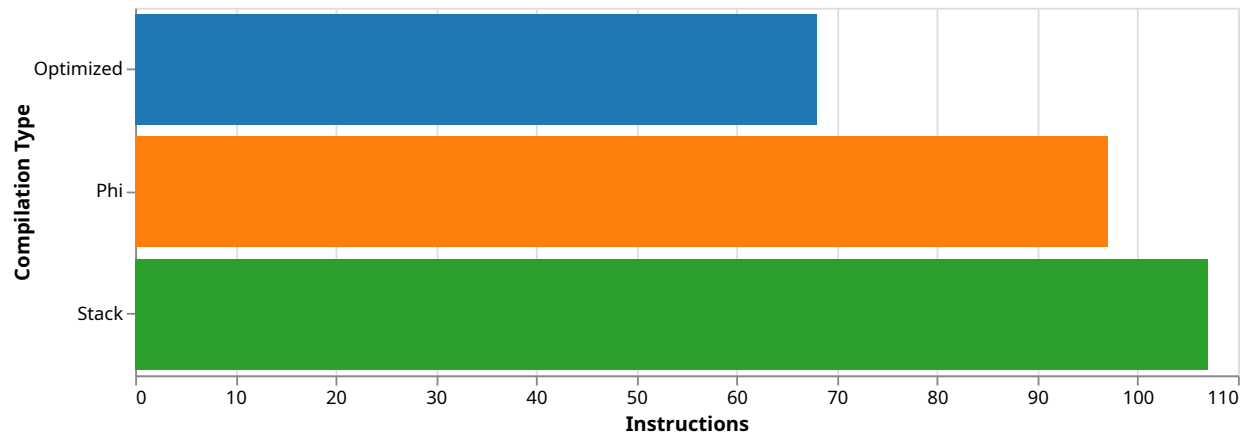
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.775253	0.0614453	10
1	Clang -O3	0.001174	1.88162e-05	10
2	Optimized	0.00163659	0.00016647	10
3	Phi	0.00172488	0.000210161	10
4	Stack	0.00156477	3.19865e-05	10



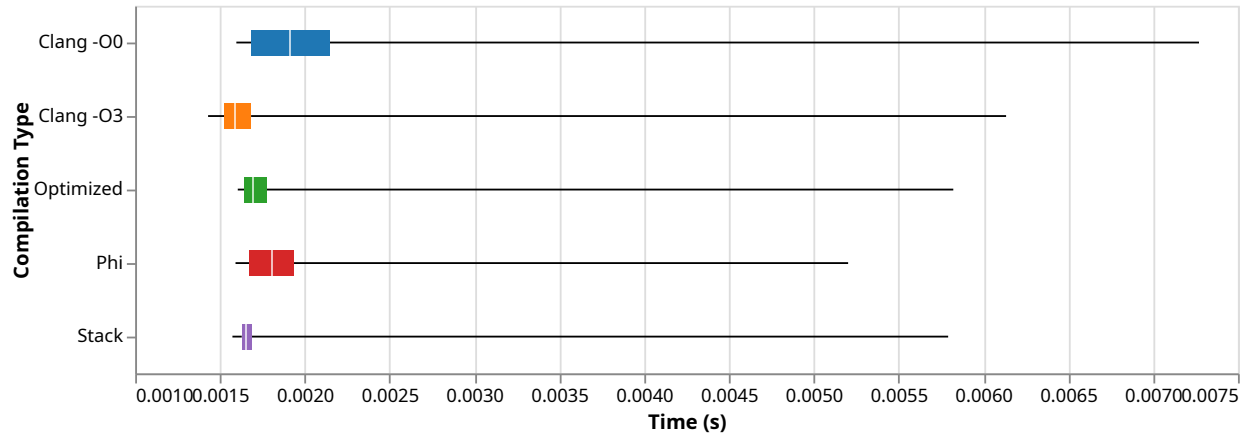
## 2.19 primes



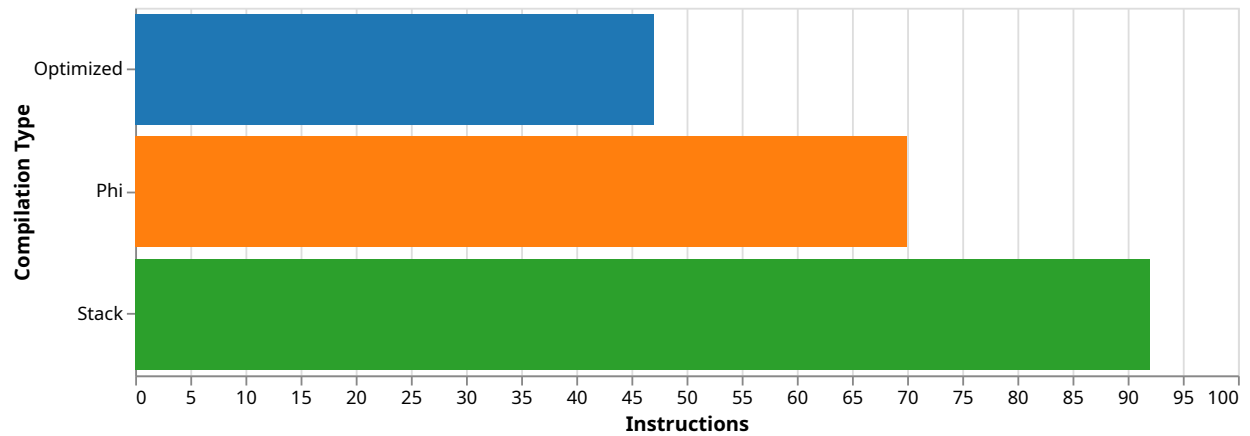
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.20195	0.00389725	10
1	Clang -O3	0.0646186	0.00162882	10
2	Optimized	0.0769258	0.00271621	10
3	Phi	0.0770291	0.00319007	10
4	Stack	0.0815229	0.00371512	10



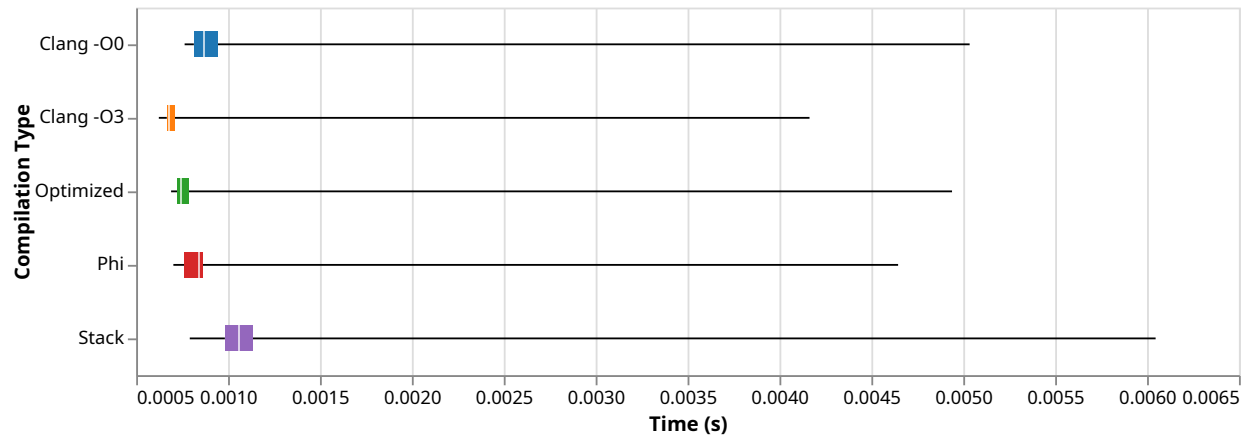
## 2.20 programBreaker



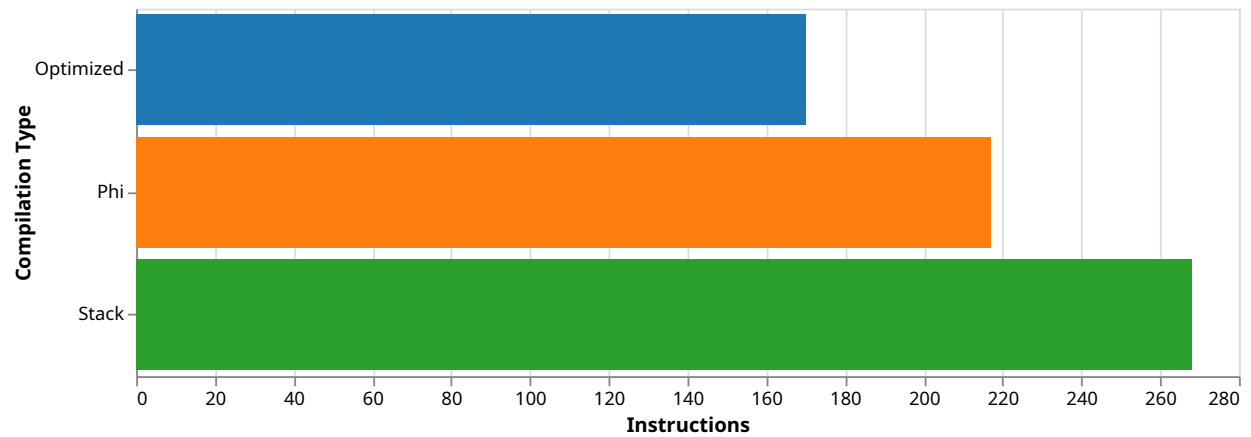
	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00212459	0.000868234	100
1	Clang -O3	0.00174558	0.000629745	100
2	Optimized	0.0017958	0.000510778	100
3	Phi	0.00196772	0.000658493	100
4	Stack	0.00172016	0.000426571	100



## 2.21 stats

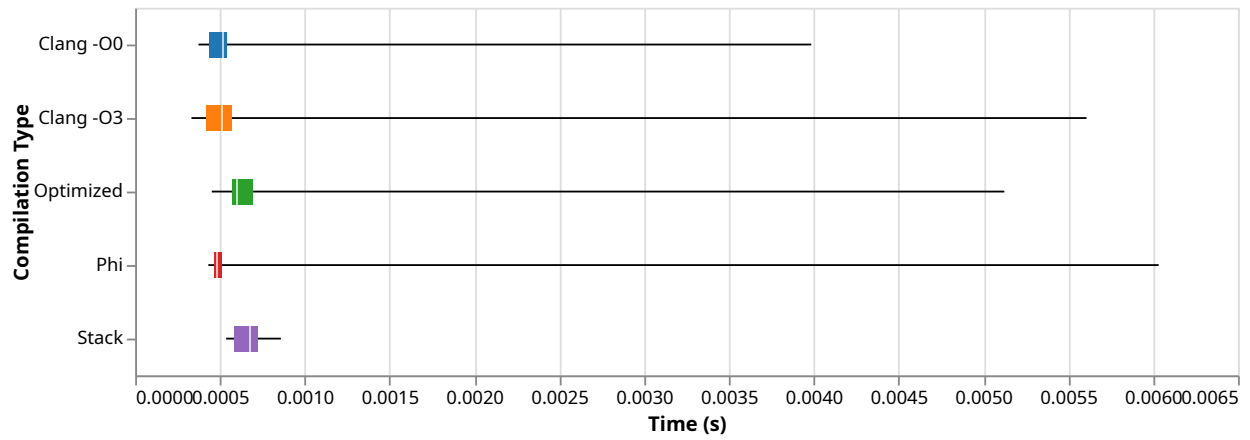


	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.00094608	0.000470526	100
1	Clang -O3	0.000767719	0.000460487	100
2	Optimized	0.000865027	0.000505186	100
3	Phi	0.000879366	0.00043165	100
4	Stack	0.00122152	0.000710277	100





## 2.22 wasteOfCycles



	Compilation Type	Average Time (s)	Standard Deviation (s)	Number of Runs
0	Clang -O0	0.000596629	0.000508627	100
1	Clang -O3	0.000691545	0.000717693	100
2	Optimized	0.00071117	0.000536161	100
3	Phi	0.000547953	0.000555762	100
4	Stack	0.000666436	9.17061e-05	100

