MINI Compiler Term Paper

Alex MacLean and Bailey Wickham

March 16, 2022

Contents

1 Introduction

Our Mini compiler meets all 6 milestones and passes all provided benchmarks with both stack- and SSA-based IR compiled to 32bit ARM assembly. We implement Sparse Simple Constant Propagation, unused SSA register elimination, function in-lining, and basic control-flow-graph simplification. Based on run-time analysis with benchmarks we see that our compiler generates code with performance comparable to `clang` without optimizations and with measurable improvements yielded by our optimizations.

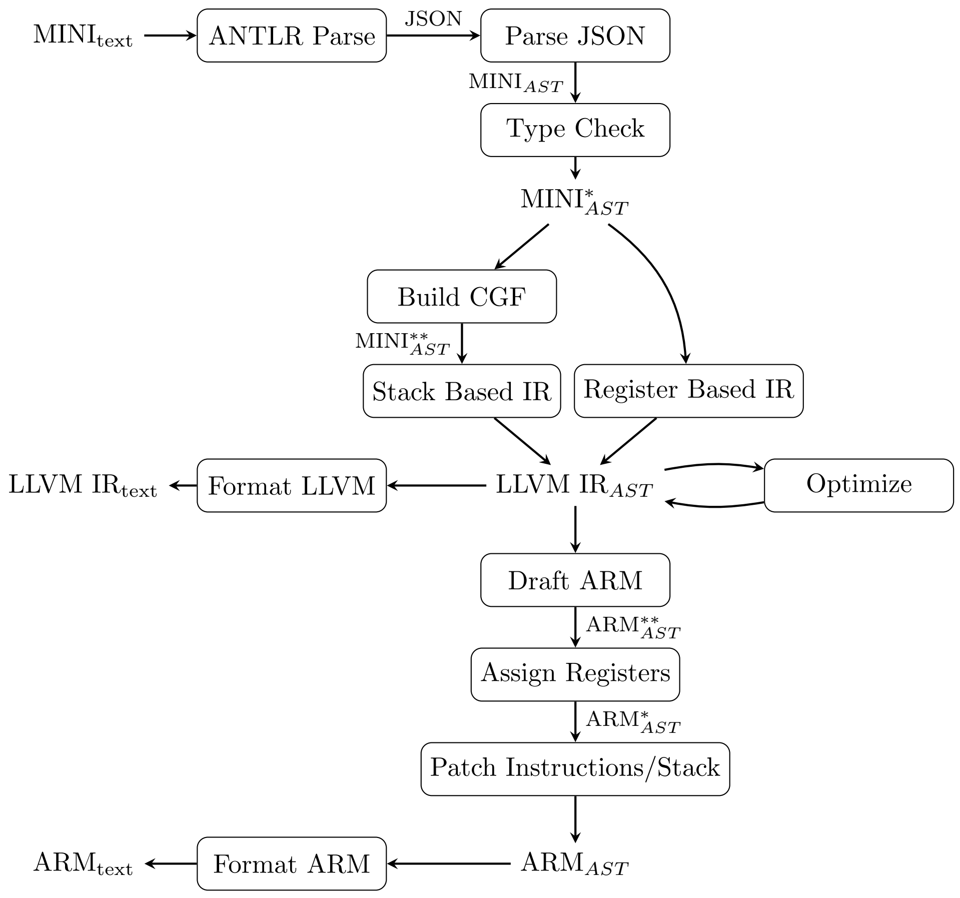

Figure 1: Overall compiler construction

2 Compiler Architecture

2.1 Parsing

The first step in our compiler is parsing the `.mini` input file. We use a sub-process to run the the given ANTLR parser to generate a JSON representation of the Mini program. This is piped into the Racket JSON library which produces hash tables that are then parsed into Racket structures. We do this by recursively traversing the hash-table JSON representation and parsing each expression and statement into a corresponding Racket structure. Some structures such as binary and unary operations were combined into a single operation struct, and empty else clauses are added, but generally the representation of the Mini AST is similar to that defined in the JSON. We make extensive use of a custom Racket match pattern to more concisely bind ids to hash tables generated from JSON. The output of the parse pass is a structure of `Mini` type which contains the types, declarations, and functions defined by the program.

2.2 Static Semantics

The type checker pass runs on the `Mini` AST provided by the parsing pass. The type checker either fails and stops compilation or returns a `Mini` structure with global variables annotated for use in the later translation passes. First global context is gathered and checked. This step includes passes over the members of structs, types of global variables, and function signatures. If correctness checks pass, this information is gathered into hash tables for use in type checking statements. We chose to support file-scope structure and function definition, allowing for mutually recursive functions and structs.

During the main type check pass, we recursively traverse functions and their bodies, verifying each statement. For control flow, we check that the guard is a boolean. For expressions we check that the input types match the expected input types. One challenge presented in future translation passes is identifying when a reference is to a global variable, especially since local declarations can hide global variables. Since this information is present in the type-checking pass we chose to annotate global variable references and assignments during this pass.

We define the condition that all non-void functions must return recursively. A block of statements always returns if at least one statement in the block will always return. `return` statements always return, `if` statements always return if the same check passes for both branches, and all other statements do not.

2.3 Intermediate Representation

The next pass of our compiler is translation into an intermediate representation: LLVM IR. Compilers us an intermediate representation to simplify the process of compiling to multiple architectures and from multiple surface languages. Additionally optimizations can all be defined as taking and producing IR so they can be safely reordered or removed.

The IR translation pass can be run in either of two modes: register or stack. Both of these passes take a `Mini` structure provided from the type checking pass and return an `LLVM` structure with the respective data management scheme. The LLVM IR is represented internally as a series of structures that correspond with their LLVM instructions ($\texttt{Store}_{\text{LL}}$, $\texttt{Call}_{\text{LL}}$, $\texttt{Binary}_{\text{LL}}$, etc...). Types are internally represented as `cons` pairs of `id` and `type`.

The first step of these passes is to translate the `Mini` into LLVM syntax. In stack mode we first directly translate the `Mini` AST into a control flow graph with Mini instructions and then execute a second pass where we translate each Mini statement to one or more IR statements. In stack mode each variable has a memory location allocated on the stack and each Mini instruction is translated to reads from memory, computation, and a write back to memory. In SSA register based mode, memory is not allocated on the stack and SSA registers are used. The Braun et. al. algorithm is employed to translate the Mini AST directly into an IR CFG.

After either pass, the program is represented as a control flow graph in the intermediate representation. `Block` structures contain lists of statements and represent a vertex in the graph, while a pair of structs $\texttt{Br}_{\text{LL}}$ and $\texttt{BrCond}_{\text{LL}}$ represent edges in the graph. The control flow graph is useful because it more closely resembles the from of the instructions the program will eventually be complied into. SSA form is used to make use-definition relations simpler and easier to analyze for register allocation and optimizations.

Additional type checking is needed during this pass. Some instructions must be truncated or zero extended to match the int size expected by the LLVM instruction. This is also handled in this pass with additional instructions being added to the blocks.

2.4 Optimizations

We implement all three required optimizations, choosing to do Sparse Simple Constant Propagation, SSA-based unused result elimination, and function in-lining, as well as some basic control flow simplification. All of these optimizations operate on the LLVM intermediate representation. We first run constant propagation using the algorithm described in the paper, traversing to detect what values are known, unknown, and unknowable, then substituting in their values. Next we do a pass to remove the unused statements. Here we reuse code from our first optimization, subtracting the set of all registers read from from the set of all registers written to get the unused registers. We then traverse the blocks, filtering unused statements. Next function in-lining is performed as described below. As a final extra optimization we perform basic control-flow graph simplification by merging blocks where the parent has one successor the a child has one predecessor. We repeatedly apply this pipeline of 4 optimizations until they can no longer optimize the program any further. This allows for constant propagation into in-lined functions and further in-lining if a function becomes eligible after other optimizations.

2.4.1 Function In-Lining

We perform function in-lining in three passes: (1) Identifying functions that can be in-lined, (2) inserting a draft version of the function blocks, (3) and repairing the CFG and ϕ-nodes. To determine whether we want to in-line a function we check if it meets the following criteria:

1. It is only called once, or it is short (where short is configurable)
2. It makes no function calls, excluding library functions such as `printf`
3. It is not `main`

We then substitute the body of the function that is being in-lined into the call site, wrapping it in a struct. We replace all references to parameters with the value of the argument and prefix all SSA registers and block labels to ensure they are unique. Finally we repair the control flow graph by updating ϕ-nodes with the new final blocks.

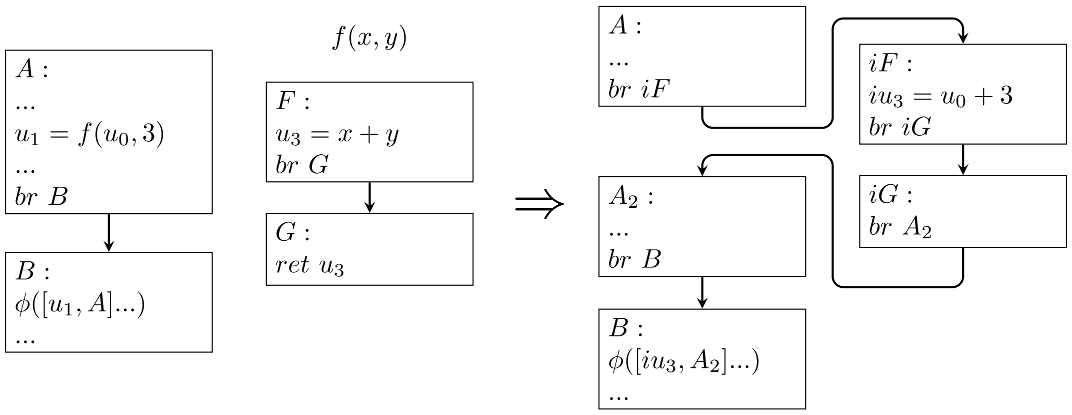

Figure 2: LLVM IR function in-lining before/after

We encountered the most errors in our compiler while implementing the in-lining phase. At first, we attempted to in-line without adding jumps, however this made resolving ϕ-nodes much more difficult. We then moved to an approach where we added a jump to the first label of the in-lined function body. This prevented us from having to rewrite the ϕ-nodes internal to the in-lined code. The merging of blocks is then performed in the subsequent cleanup pass.

2.5 ARM Code Generation and Register Allocation

ARM translation occurs in several steps: First we draft a rough version of ARM with infinite registers and placeholder stack locations, next we assign registers, and finally we cleanup the ARM and assign stack locations. `draft-arm` does the bulk of the direct translation, translating LLVM-IR statements to their respective ARM instructions, adding instructions for saving and loading function call parameters, and handling ϕ-statements. ϕ-statements exist in the LLVM-IR and must be translated into ARM instructions in the draft phase. Since ϕ-statements do not correspond to any real ARM instruction they must be removed. Our compiler does this by adding moves of the value into the ϕ register to the bottom of each block the ϕ-node references.

This draft is then revised to produce valid ARM instructions. First register allocation is performed. This step performs live variable analysis to build a conflict graph, adding an edge between any two registers that are live at the same time. Next the coloring of this graph is used to assign registers and spill locations. Next, final stack locations are assigned and substituted and the headers and footers of the function are added. In a final pass, we perform some very minor optimizations and cleanup, such as removing moves with the same source and target.

3 Racket

For this project we chose to implement our compiler in un-typed Racket. The choice of Racket presented both some interesting challenges and opportunities. Over the course of the term we developed several useful macros and design patterns.

3.1 Mutli-function Closures

One common issue we encountered was wanting to recursively traverse the program structure while maintaining some context information. One of the first places this was required was type-checking, where we wanted to keep structure and function signature information as we checked each statement and expression. To accomplish this we define a wrapper function `check-stmt*` which contains function specific parameters like the current type environment. We then define inner functions within the wrapper such as `check-stmt` and `check-expression`, which use the values the outer function is closed over (like the type environment). This pattern is similar to object oriented programming, however it retains many of the benefits of functional programming. In this example, the `check-stmt*` acts as an object bound to the current functions type environment, with each function generating its own `check-stmt*`. The return type of `check-stmt*` is a function, which when called performs the verification. This pattern allows us to maintain context while recursively traversing a data structure without needing to thread numerous parameters through every call site.

3.2 Matching on Parameters

Another Racket feature we made repeated use of was macros. We defined a pair of macros `define+` and $\lambda+$. This are used to put match patterns in function definitions. For example:

```
(define (add-cons (cons a b))
  (+ a b))
```

defines a function which takes a pair and returns the sum of the elements. These macros proved very useful in cases across the program where we wanted to accept a struct and use its members without the clunky struct accessor methods.

4 Metrics

We ran our metrics on the 32bit Raspberry Pi provided to our class. We first generated the assembly locally since Racket is unsupported on ARM. We then transferred these files to the Pi and for final assembly into ARM. The script `run_benchmarks.sh` runs each of the benchmarks for each type of compilation 10 times and records the runtimes in a CSV. For each benchmark, we count the instructions of our generated code. We ignore labels and function headers, only counting instructions.

4.1 BenchMarkishTopics

	LLVM IR	ARM Assembly
Stack	130	204
Registers	82	184
Optimizations	74	165

Table 1: BenchMarkishTopics instruction counts for generated code excluding labels

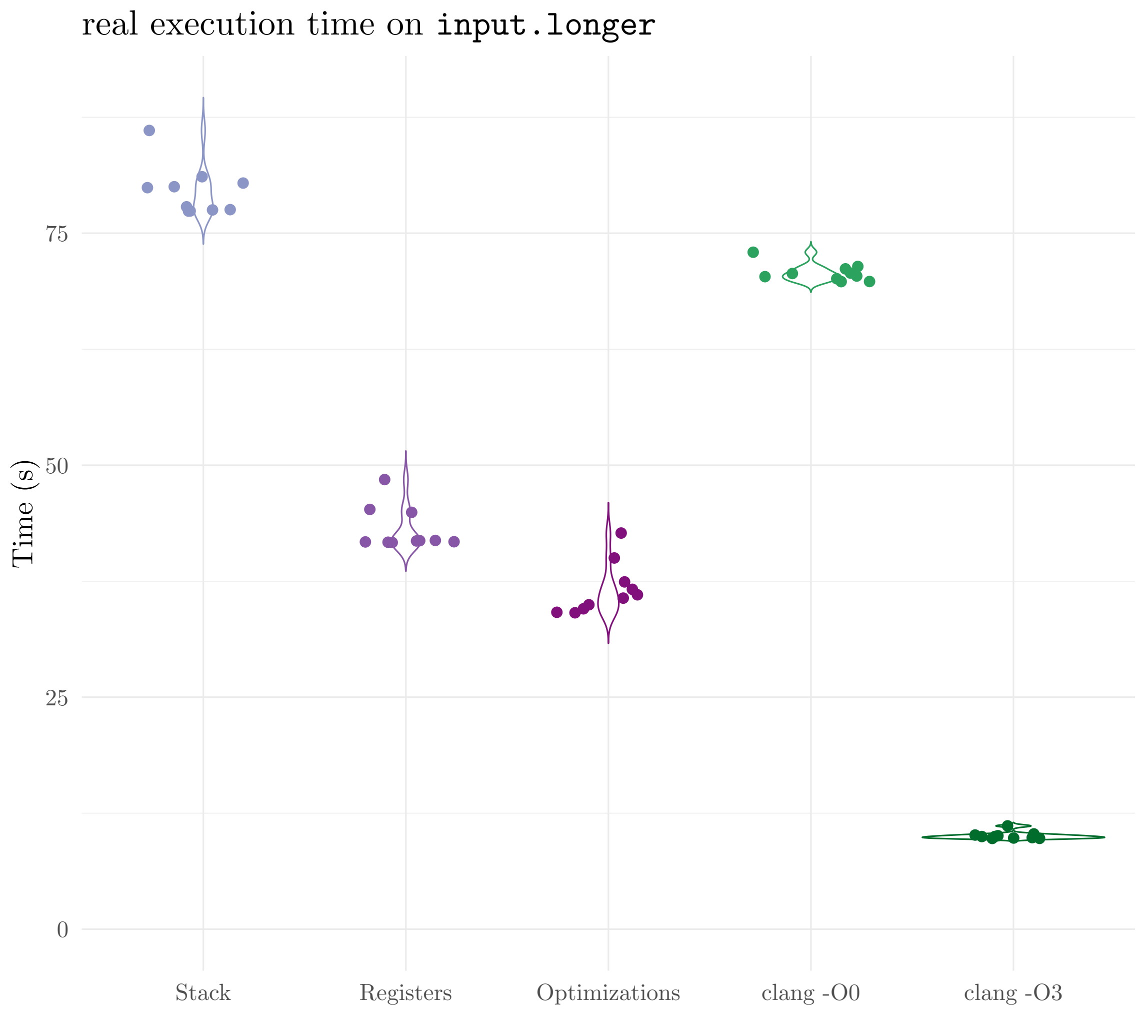

Figure 3: BenchMarkishTopics runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	77.38	77.52	78.88	79.52	80.31	86.08
Registers	41.67	41.74	41.85	43.10	44.16	48.45
Optimizations	34.09	34.63	35.86	36.62	37.22	42.69
clang -O0	69.78	70.15	70.52	70.73	71.05	72.95
clang -O3	9.78	9.83	9.98	10.08	10.13	11.13

Table 2: BenchMarkishTopics runtime (in seconds) summary statistic based on 10 runs each

4.2 Fibonacci

	LLVM IR	ARM Assembly
Stack	36	68
Registers	20	53
Optimizations	19	52

Table 3: Fibonacci instruction counts for generated code excluding labels

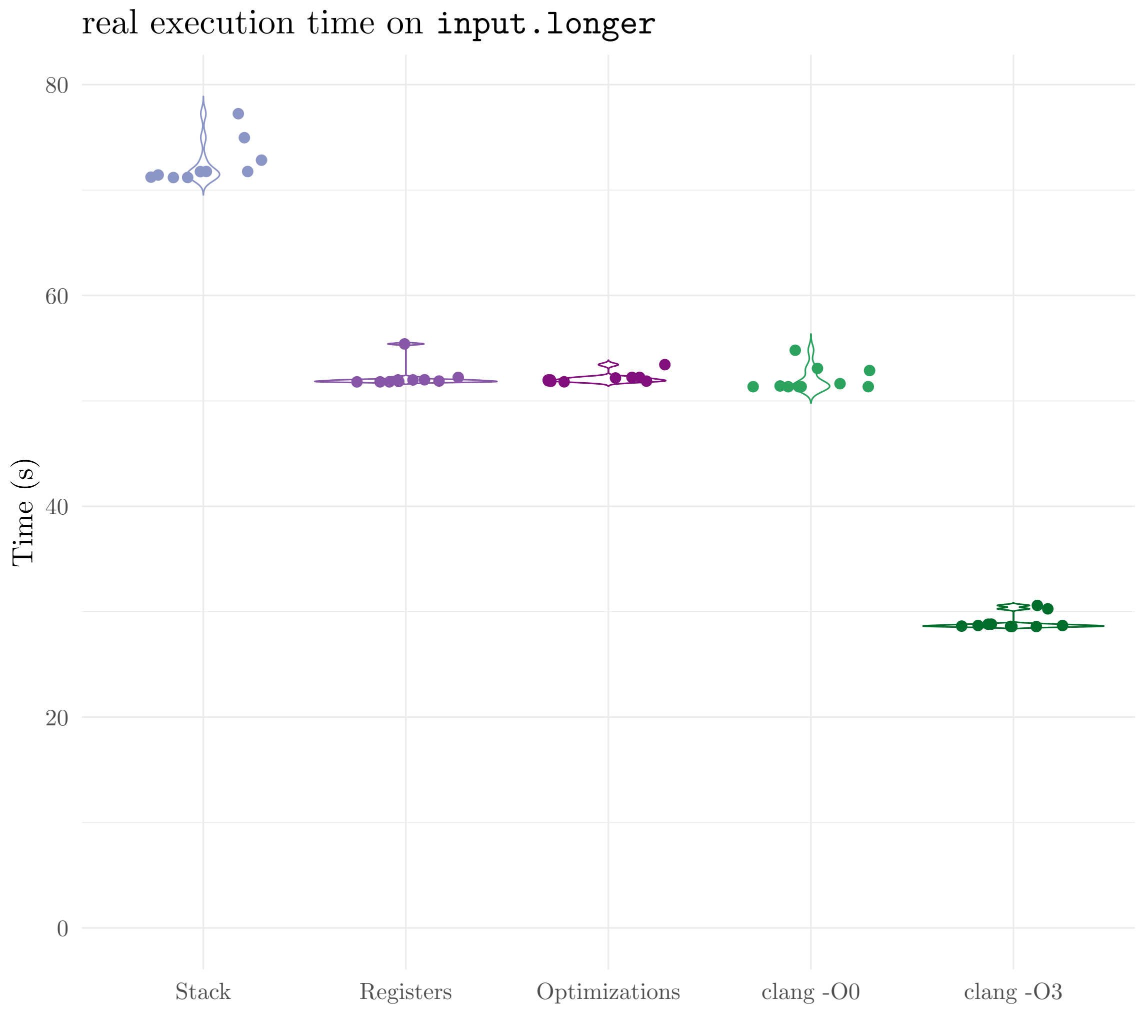

Figure 4: Fibonacci runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	71.19	71.28	71.75	72.54	72.57	77.25
Registers	51.80	51.82	51.93	52.28	52.00	55.40
Optimizations	51.81	51.88	51.99	52.15	52.22	53.44
clang -O0	51.34	51.35	51.38	52.05	52.57	54.80
clang -O3	28.59	28.61	28.69	29.03	28.82	30.59

Table 4: Fibonacci runtime (in seconds) summary statistic based on 10 runs each

4.3 GeneralFunctAndOptimize

	LLVM IR	ARM Assembly
Stack	147	221
Registers	104	227
Optimizations	84	185

Table 5: GeneralFunctAndOptimize instruction counts for generated code excluding labels

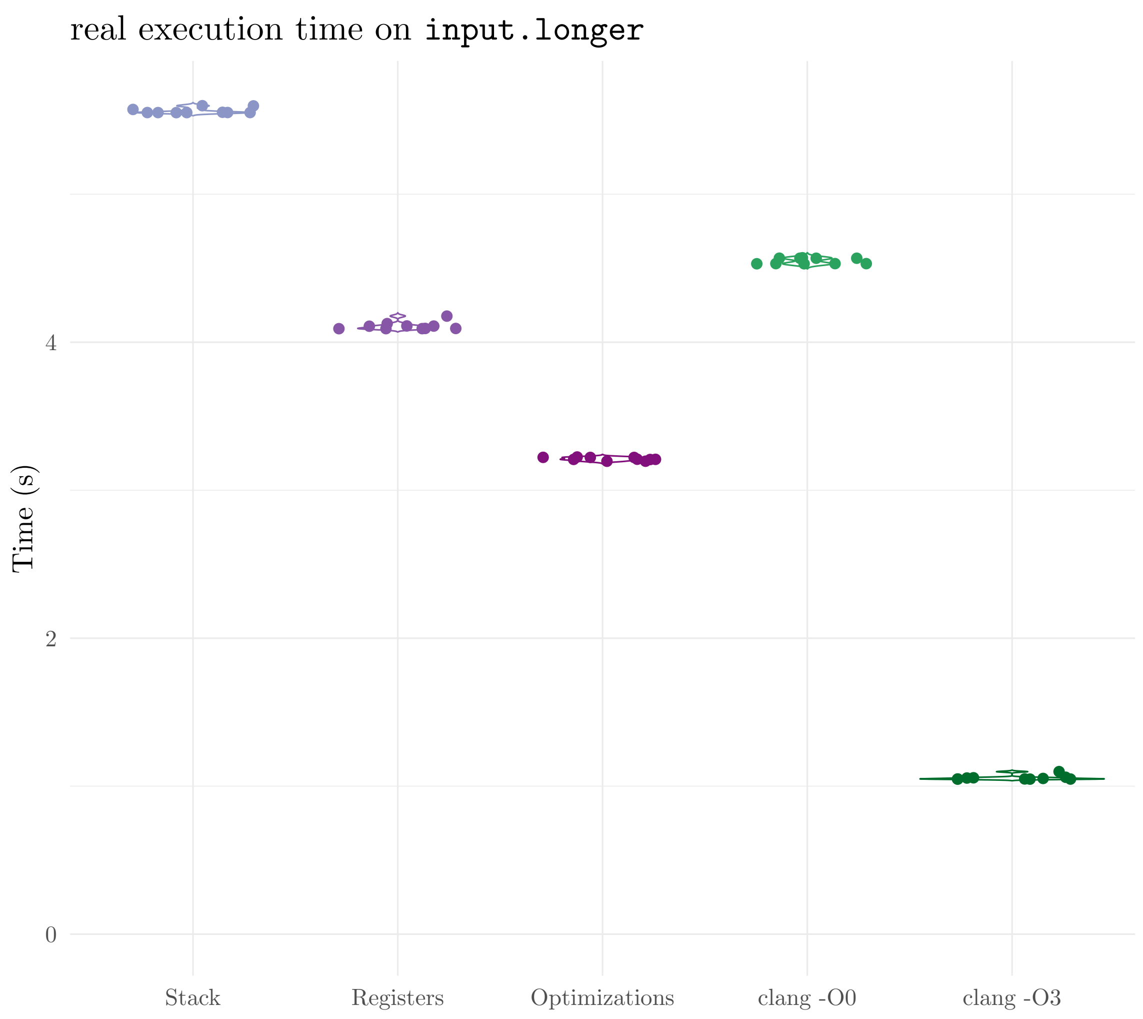

Figure 5: GeneralFunctAndOptimize runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	5.55	5.55	5.55	5.56	5.57	5.60
Registers	4.09	4.09	4.10	4.11	4.11	4.18
Optimizations	3.20	3.21	3.21	3.21	3.22	3.22
clang -O0	4.53	4.53	4.55	4.55	4.57	4.57
clang -O3	1.05	1.05	1.05	1.06	1.06	1.10

Table 6: GeneralFunctAndOptimize runtime (in seconds) summary statistic based on 10 runs each

4.4 OptimizationBenchmark

	LLVM IR	ARM Assembly
Stack	1045	1380
Registers	418	1155
Optimizations	152	398

Table 7: OptimizationBenchmark instruction counts for generated code excluding labels

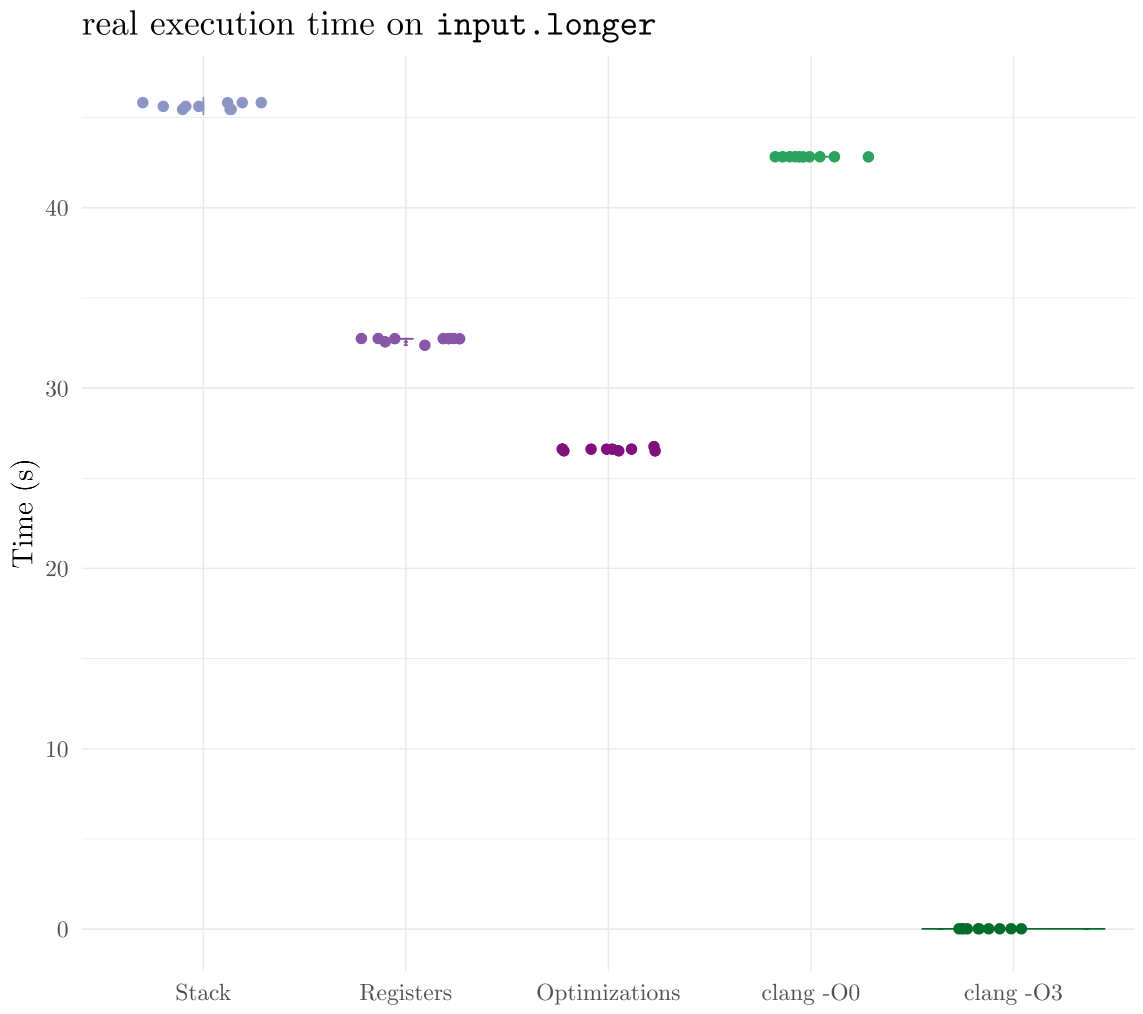

Figure 6: OptimizationBenchmark runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	45.45	45.49	45.62	45.65	45.82	45.83
Registers	32.38	32.73	32.74	32.69	32.75	32.75
Optimizations	26.52	26.54	26.61	26.60	26.62	26.76
clang -O0	42.82	42.82	42.83	42.82	42.83	42.83
clang -O3	0.02	0.02	0.02	0.02	0.02	0.02

Table 8: OptimizationBenchmark runtime (in seconds) summary statistic based on 10 runs each

4.5 TicTac

	LLVM IR	ARM Assembly
Stack	464	739
Registers	390	667
Optimizations	358	619

Table 9: TicTac instruction counts for generated code excluding labels

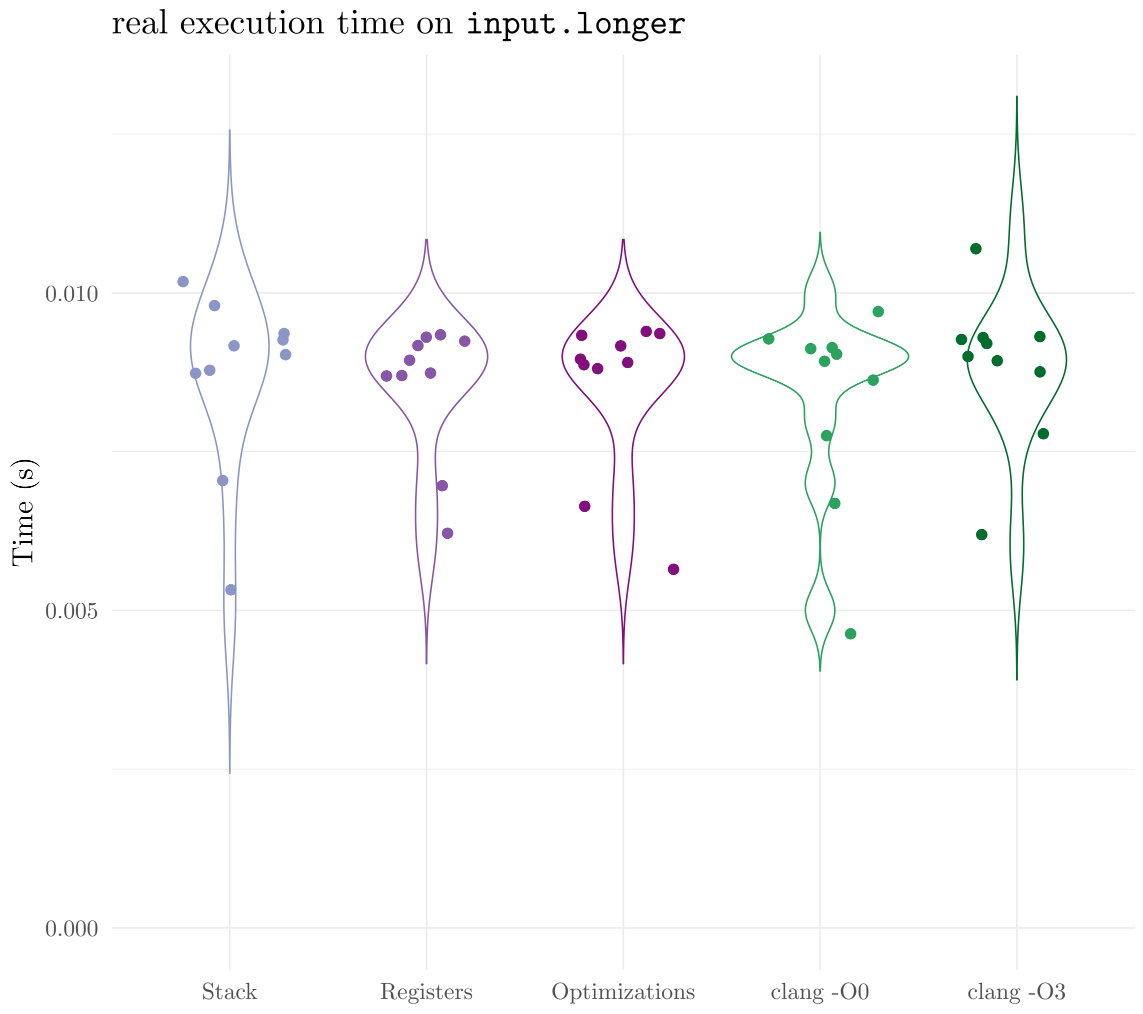

Figure 7: TicTac runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.00	0.01	0.01	0.01	0.01	0.01
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.01	0.01	0.01
clang -O0	0.00	0.01	0.01	0.01	0.01	0.01
clang -O3	0.01	0.01	0.01	0.01	0.01	0.01

Table 10: TicTac runtime (in seconds) summary statistic based on 10 runs each

4.6 bert

	LLVM IR	ARM Assembly
Stack	694	1152
Registers	432	1037
Optimizations	382	950

Table 11: bert instruction counts for generated code excluding labels

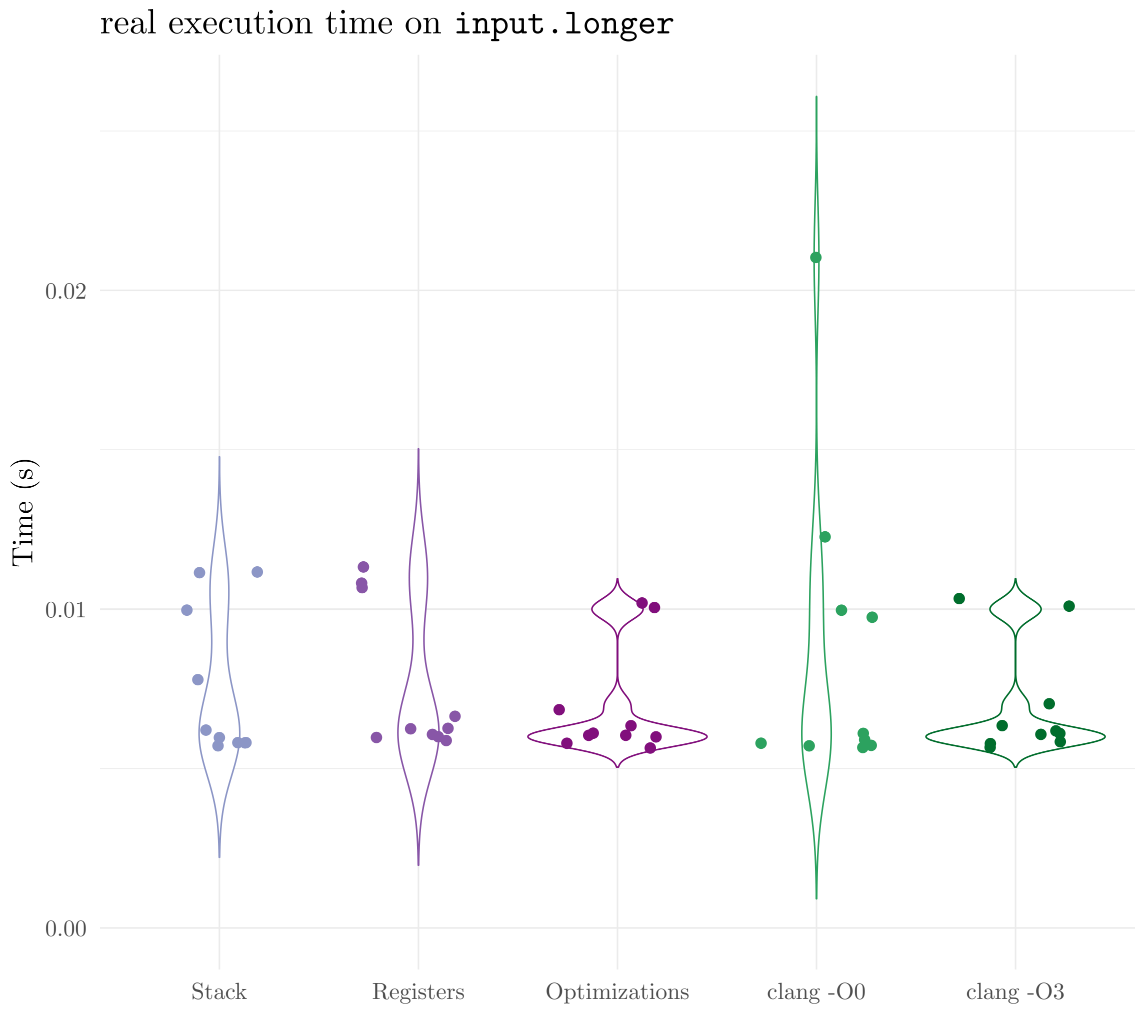

Figure 8: bert runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.01	0.01	0.01	0.01	0.01	0.01
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.01	0.01	0.01
clang -O0	0.01	0.01	0.01	0.01	0.01	0.02
clang -O3	0.01	0.01	0.01	0.01	0.01	0.01

Table 12: bert runtime (in seconds) summary statistic based on 10 runs each

4.7 biggest

	LLVM IR	ARM Assembly
Stack	97	131
Registers	51	107
Optimizations	44	87

Table 13: biggest instruction counts for generated code excluding labels

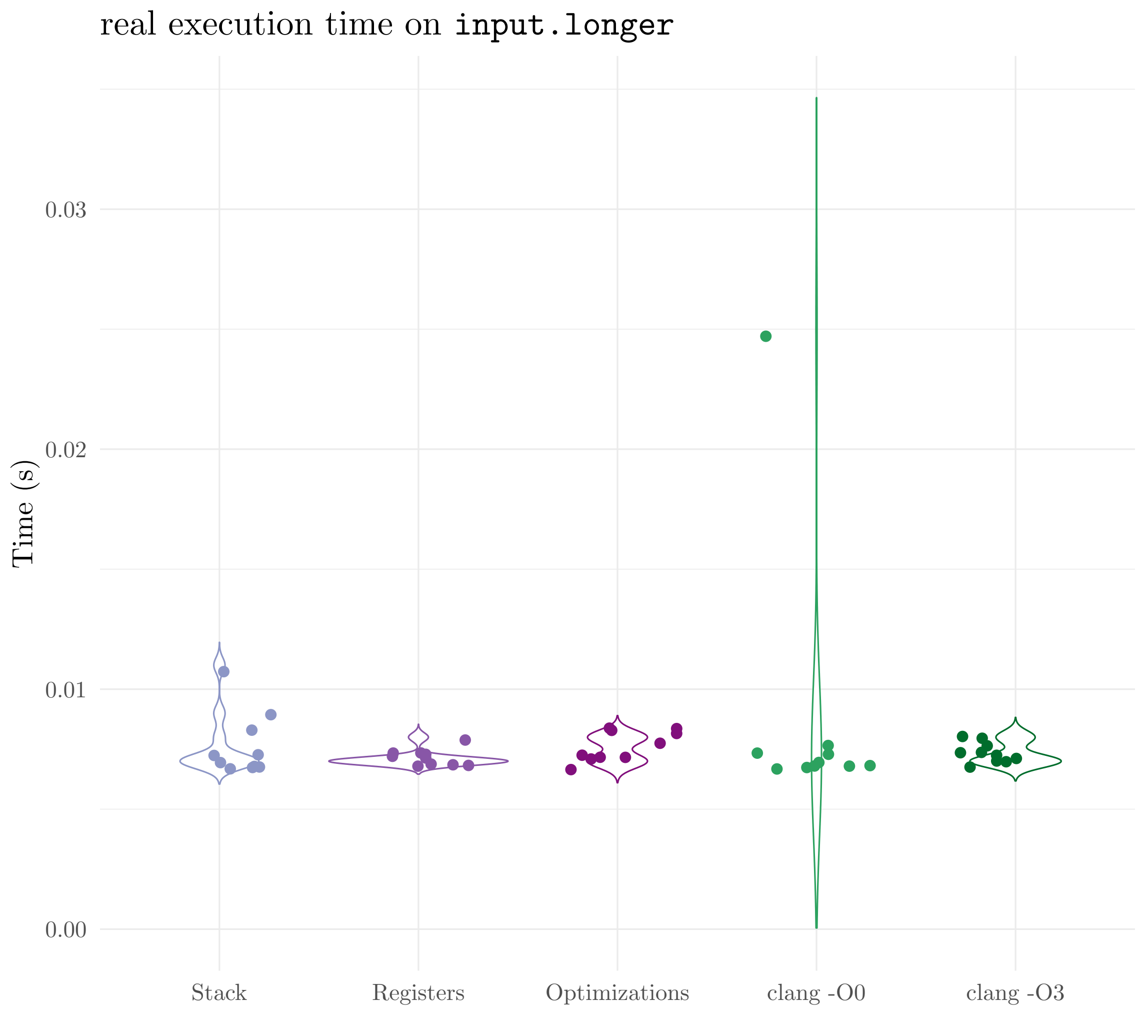

Figure 9: biggest runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.01	0.01	0.01	0.01	0.01	0.01
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.01	0.01	0.01
clang -O0	0.01	0.01	0.01	0.01	0.01	0.02
clang -O3	0.01	0.01	0.01	0.01	0.01	0.01

Table 14: biggest runtime (in seconds) summary statistic based on 10 runs each

4.8 binaryConverter

	LLVM IR	ARM Assembly
Stack	131	198
Registers	61	148
Optimizations	52	127

Table 15: binaryConverter instruction counts for generated code excluding labels

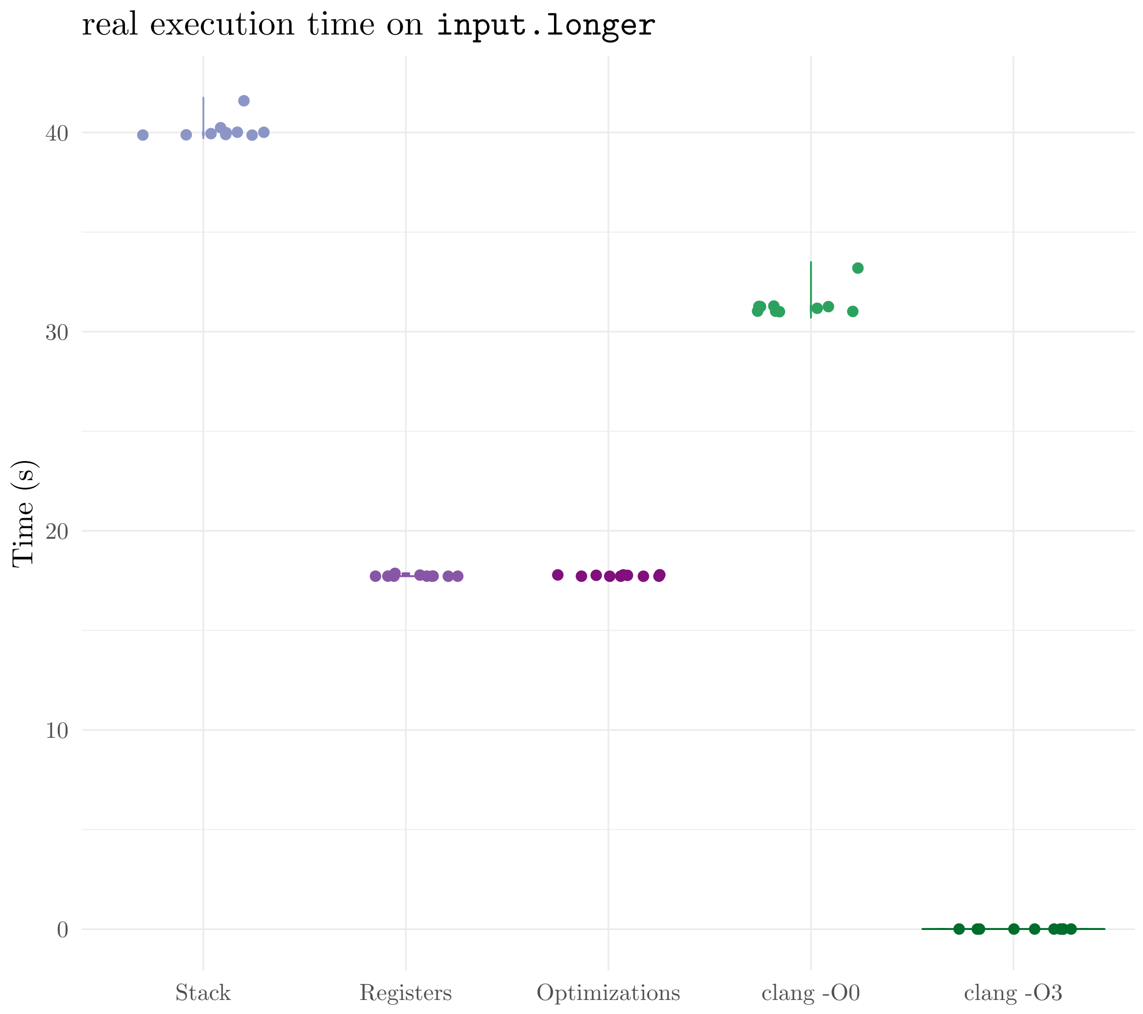

Figure 10: binaryConverter runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	39.87	39.89	39.97	40.13	40.01	41.59
Registers	17.72	17.72	17.73	17.74	17.73	17.86
Optimizations	17.72	17.72	17.74	17.75	17.78	17.80
clang -O0	31.00	31.03	31.22	31.35	31.27	33.19
clang -O3	0.00	0.00	0.01	0.01	0.01	0.01

Table 16: binaryConverter runtime (in seconds) summary statistic based on 10 runs each

4.9 brett

	LLVM IR	ARM Assembly
Stack	715	1391
Registers	581	1295
Optimizations	477	1125

Table 17: brett instruction counts for generated code excluding labels

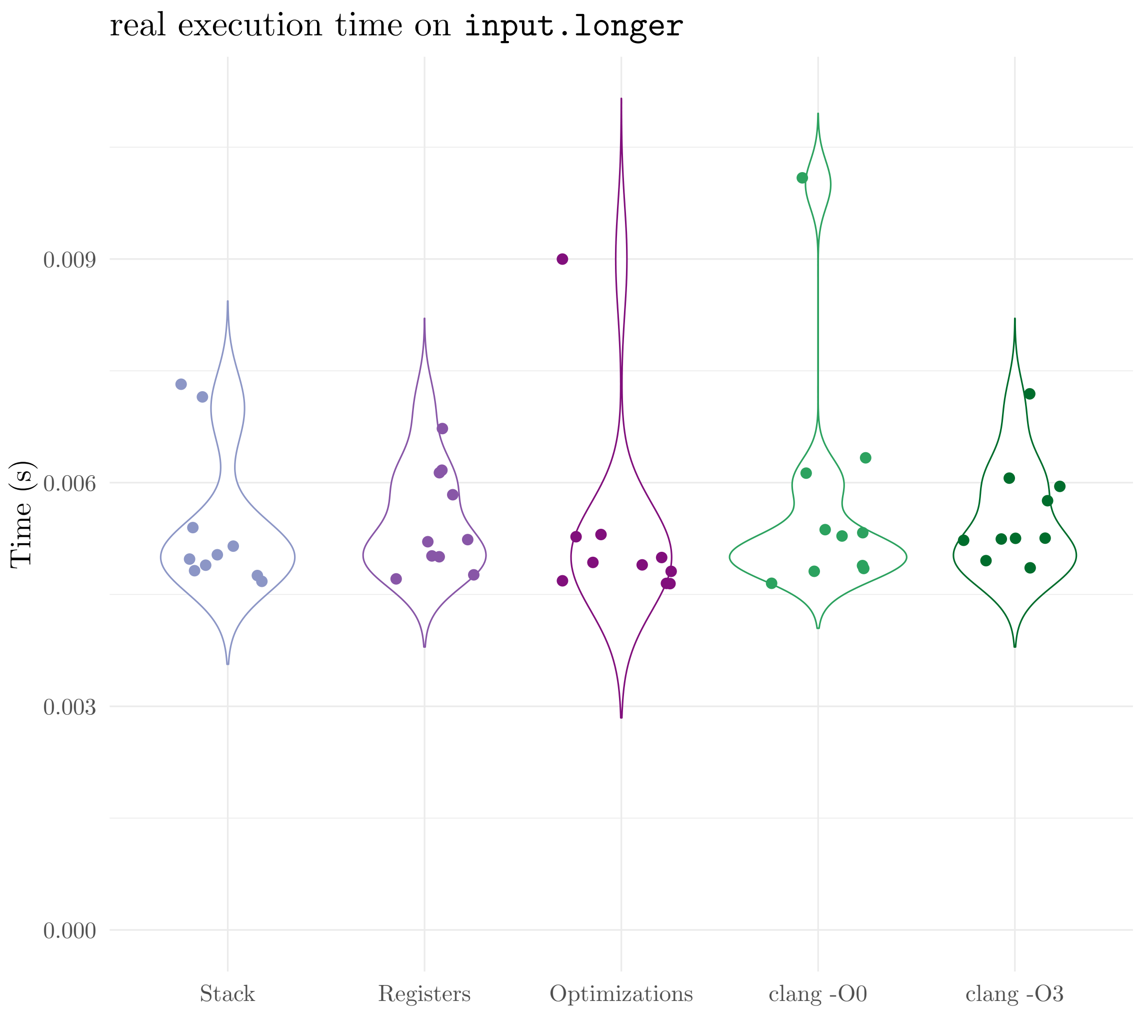

Figure 11: brett runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.00	0.00	0.00	0.01	0.00	0.01
Registers	0.00	0.00	0.00	0.01	0.01	0.01
Optimizations	0.00	0.00	0.00	0.01	0.00	0.01
clang -O0	0.00	0.00	0.00	0.01	0.01	0.01
clang -O3	0.00	0.00	0.00	0.01	0.01	0.01

Table 18: brett runtime (in seconds) summary statistic based on 10 runs each

4.10 creativeBenchMarkName

	LLVM IR	ARM Assembly
Stack	240	314
Registers	140	293
Optimizations	127	245

Table 19: creativeBenchMarkName instruction counts for generated code excluding labels

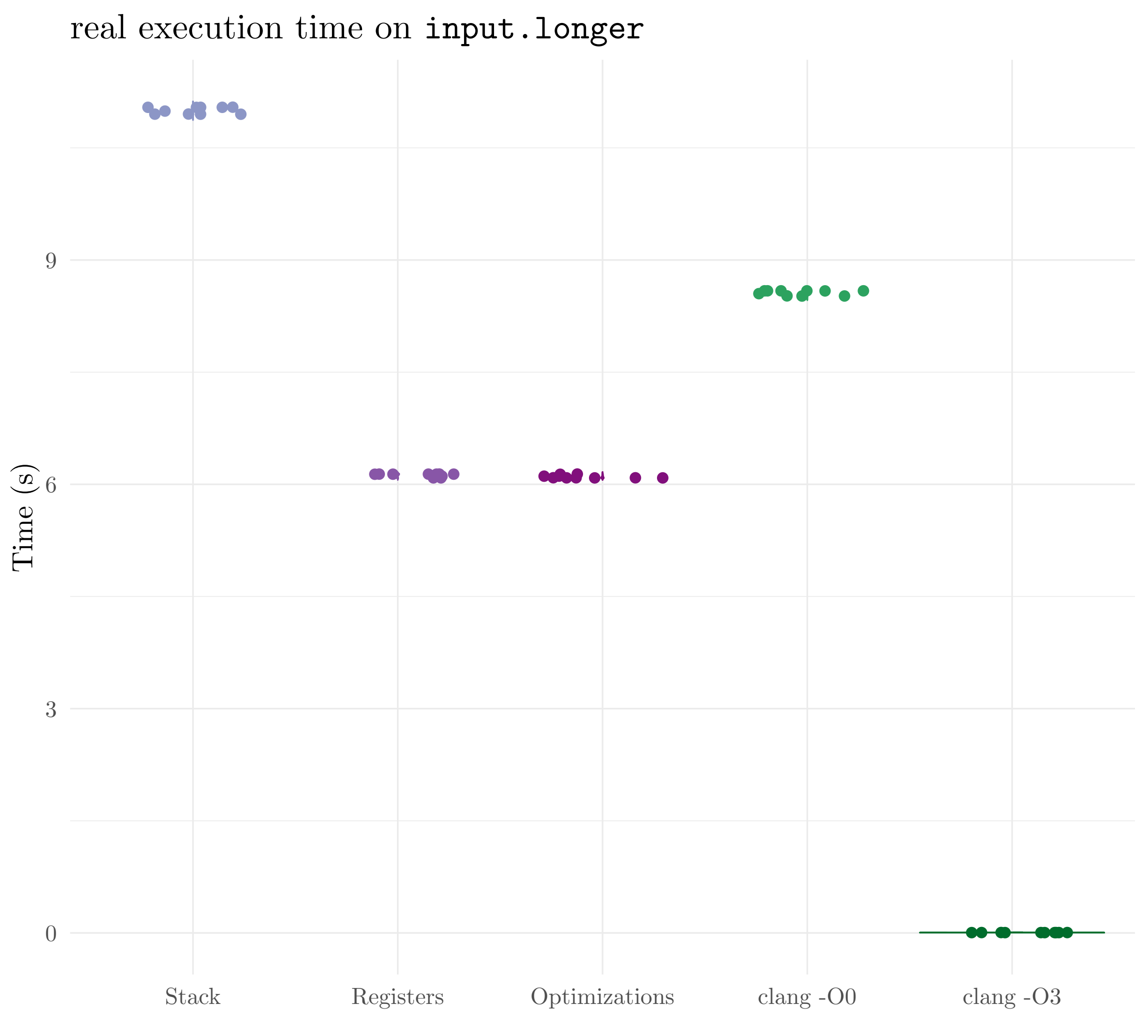

Figure 12: creativeBenchMarkName runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	10.95	10.95	11.02	11.00	11.04	11.04
Registers	6.09	6.12	6.14	6.12	6.14	6.14
Optimizations	6.09	6.09	6.09	6.10	6.11	6.14
clang -O0	8.52	8.53	8.59	8.56	8.59	8.59
clang -O3	0.00	0.00	0.00	0.01	0.00	0.01

Table 20: creativeBenchMarkName runtime (in seconds) summary statistic based on 10 runs each

4.11 fact_sum

	LLVM IR	ARM Assembly
Stack	74	127
Registers	40	107
Optimizations	30	90

Table 21: fact_sum instruction counts for generated code excluding labels

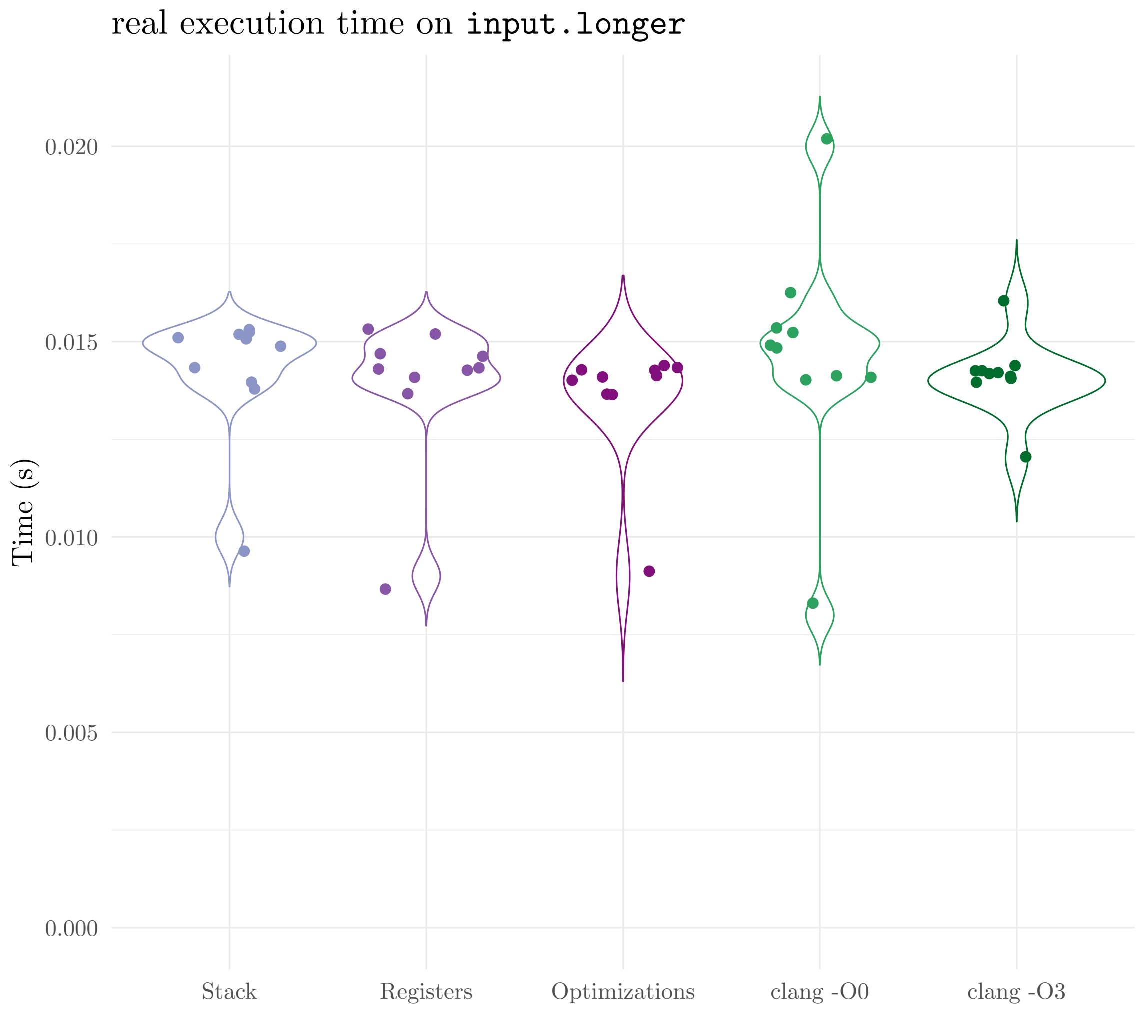

Figure 13: fact_sum runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.01	0.01	0.01	0.01	0.01	0.01
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.01	0.01	0.01
clang -O0	0.01	0.01	0.01	0.01	0.01	0.02
clang -O3	0.01	0.01	0.01	0.01	0.01	0.02

Table 22: fact_sum runtime (in seconds) summary statistic based on 10 runs each

4.12 hailstone

	LLVM IR	ARM Assembly
Stack	52	94
Registers	30	80
Optimizations	23	60

Table 23: hailstone instruction counts for generated code excluding labels

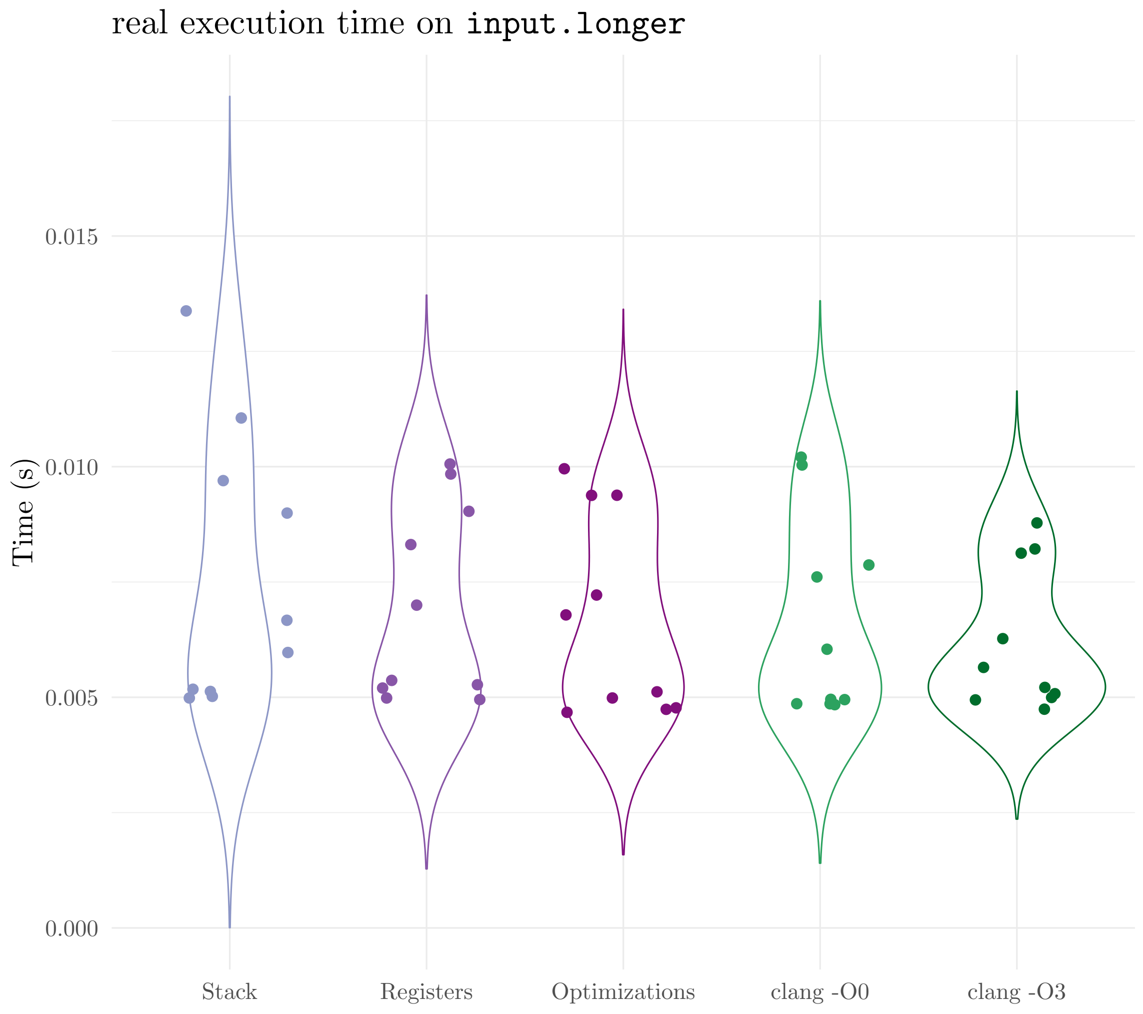

Figure 14: hailstone runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.00	0.00	0.01	0.01	0.01	0.01
Registers	0.00	0.00	0.01	0.01	0.01	0.01
Optimizations	0.00	0.00	0.01	0.01	0.01	0.01
clang -O0	0.00	0.00	0.01	0.01	0.01	0.01
clang -O3	0.00	0.00	0.01	0.01	0.01	0.01

Table 24: hailstone runtime (in seconds) summary statistic based on 10 runs each

4.13 hanoi_benchmark

	LLVM IR	ARM Assembly
Stack	198	364
Registers	138	339
Optimizations	134	335

Table 25: hanoi_benchmark instruction counts for generated code excluding labels

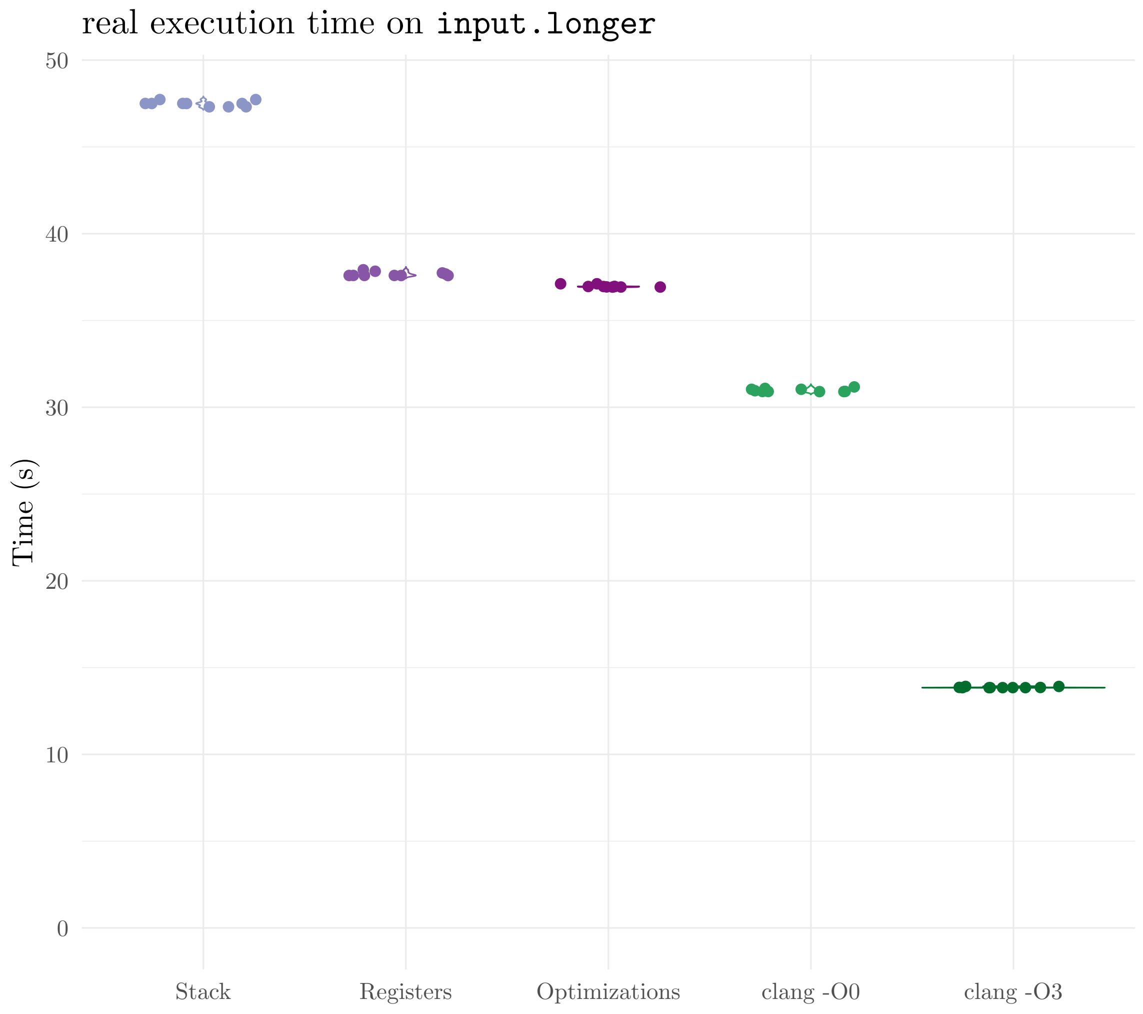

Figure 15: hanoi_benchmark runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	47.31	47.36	47.50	47.49	47.50	47.73
Registers	37.59	37.59	37.59	37.67	37.72	37.92
Optimizations	36.92	36.93	36.95	36.98	36.96	37.11
clang -O0	30.90	30.91	30.93	30.98	31.04	31.17
clang -O3	13.84	13.85	13.85	13.86	13.86	13.92

Table 26: hanoi_benchmark runtime (in seconds) summary statistic based on 10 runs each

4.14 killerBubbles

	LLVM IR	ARM Assembly
Stack	193	252
Registers	117	216
Optimizations	104	191

Table 27: killerBubbles instruction counts for generated code excluding labels

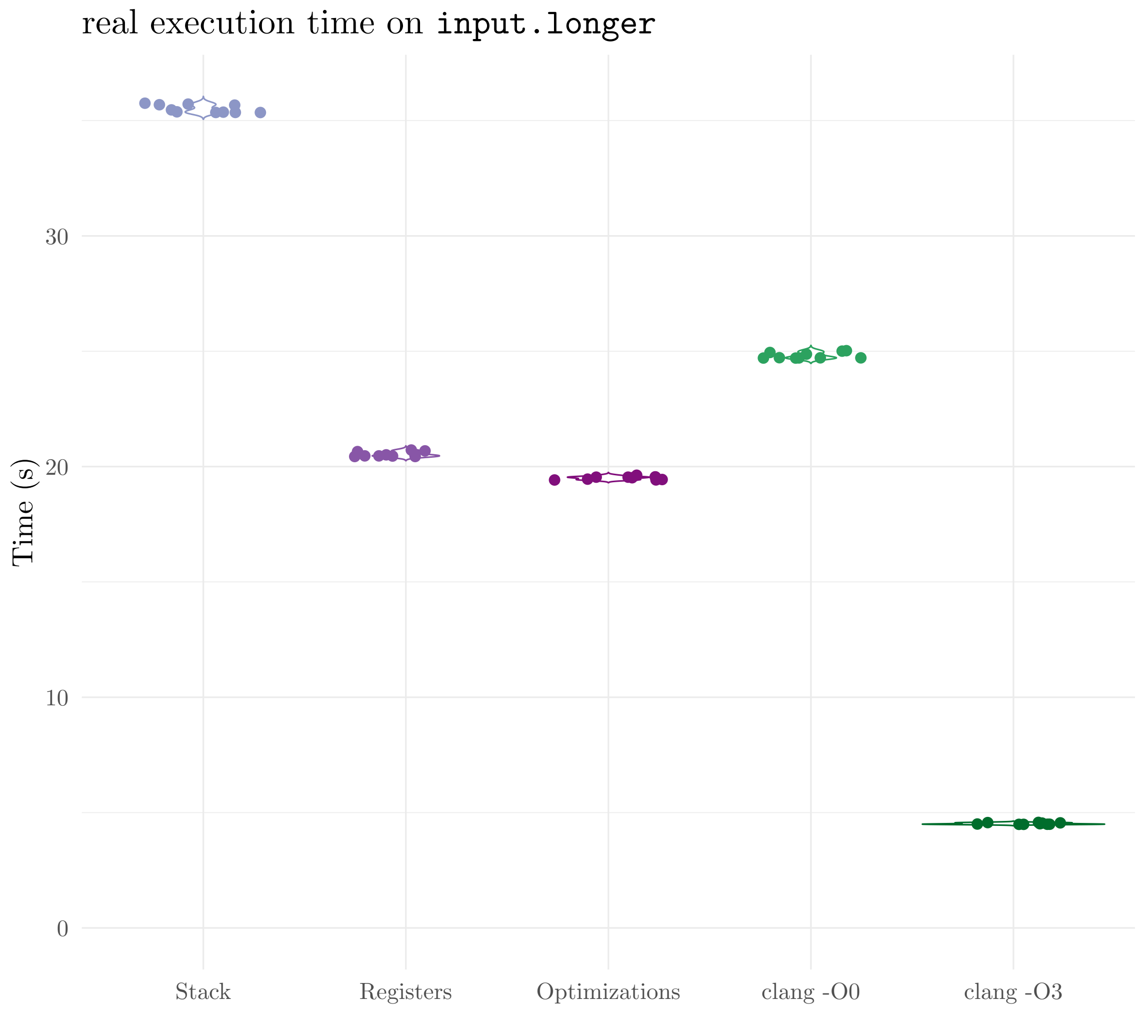

Figure 16: killerBubbles runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	35.35	35.36	35.42	35.51	35.68	35.75
Registers	20.44	20.46	20.48	20.53	20.61	20.72
Optimizations	19.42	19.44	19.53	19.51	19.55	19.63
clang -O0	24.70	24.71	24.72	24.81	24.93	25.02
clang -O3	4.49	4.50	4.51	4.52	4.55	4.58

Table 28: killerBubbles runtime (in seconds) summary statistic based on 10 runs each

4.15 mile1

	LLVM IR	ARM Assembly
Stack	75	115
Registers	44	101
Optimizations	38	86

Table 29: mile1 instruction counts for generated code excluding labels

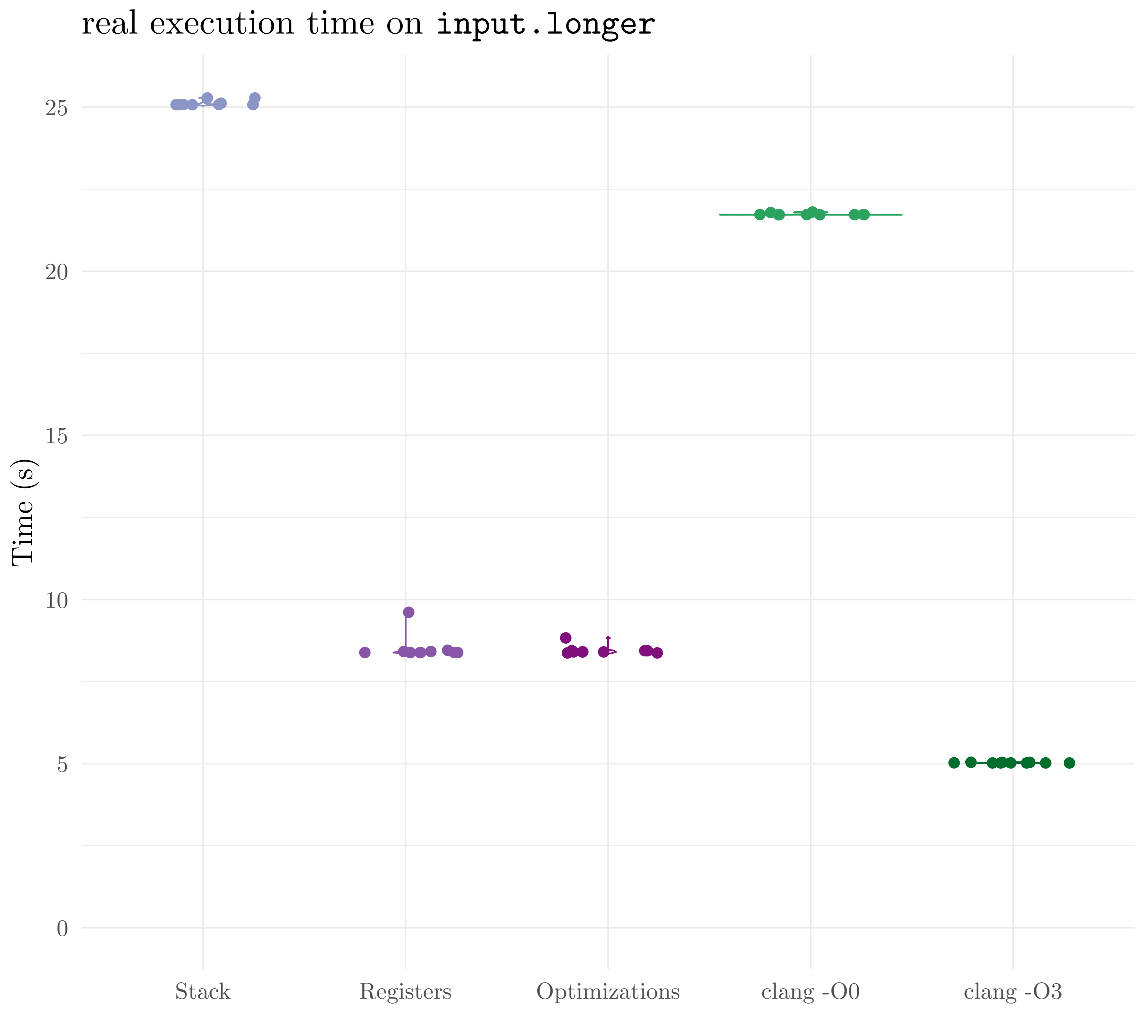

Figure 17: mile1 runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	25.07	25.08	25.08	25.12	25.11	25.28
Registers	8.38	8.38	8.38	8.52	8.42	9.61
Optimizations	8.37	8.40	8.40	8.45	8.44	8.83
clang -O0	21.72	21.72	21.73	21.74	21.73	21.80
clang -O3	5.02	5.02	5.02	5.03	5.03	5.04

Table 30: mile1 runtime (in seconds) summary statistic based on 10 runs each

4.16 mixed

	LLVM IR	ARM Assembly
Stack	215	286
Registers	137	257
Optimizations	100	204

Table 31: mixed instruction counts for generated code excluding labels

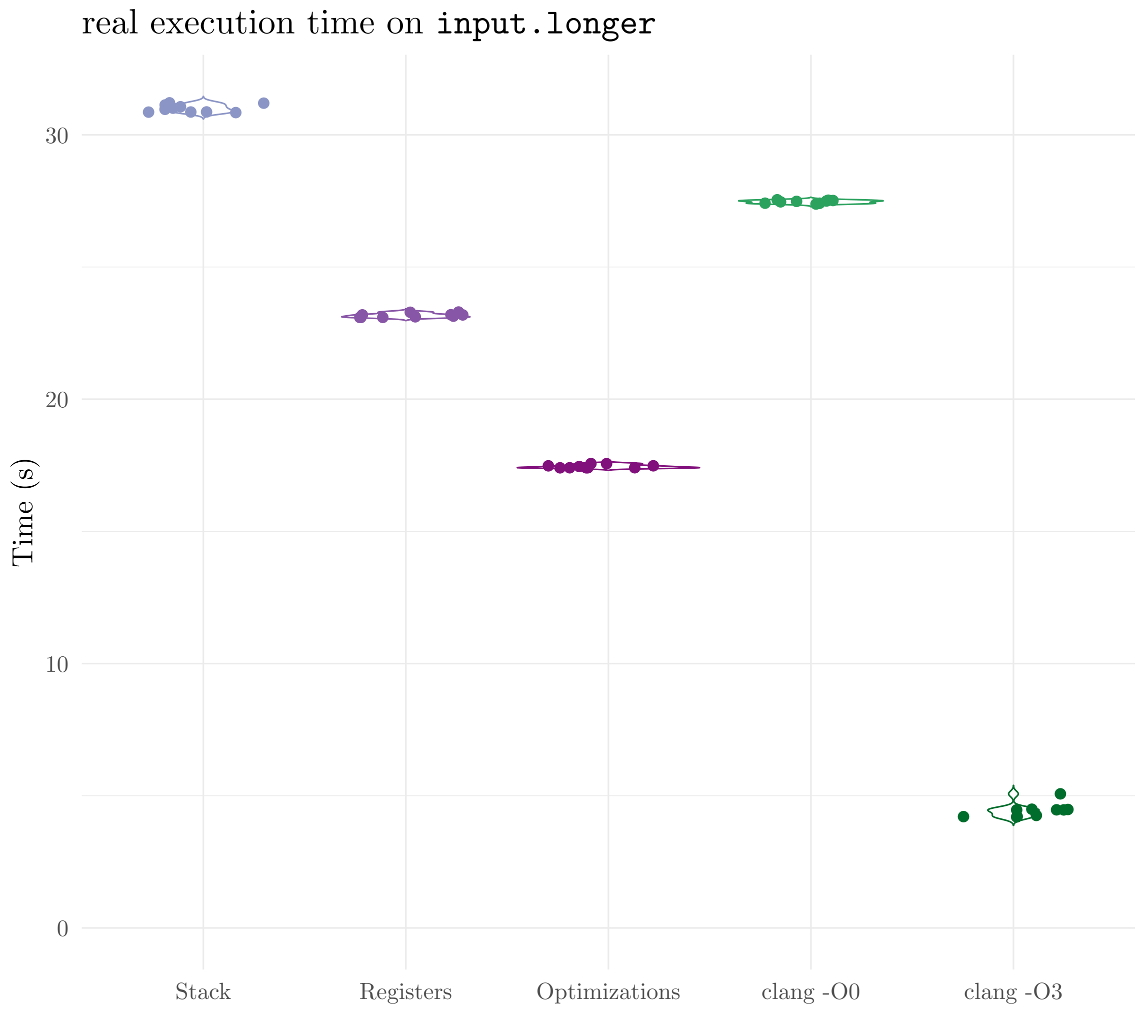

Figure 18: mixed runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	30.84	30.86	30.99	31.00	31.11	31.21
Registers	23.09	23.10	23.16	23.17	23.19	23.30
Optimizations	17.41	17.41	17.43	17.46	17.48	17.57
clang -O0	27.38	27.41	27.47	27.46	27.51	27.54
clang -O3	4.20	4.22	4.46	4.43	4.48	5.07

Table 32: mixed runtime (in seconds) summary statistic based on 10 runs each

4.17 primes

	LLVM IR	ARM Assembly
Stack	103	162
Registers	58	136
Optimizations	48	116

Table 33: primes instruction counts for generated code excluding labels

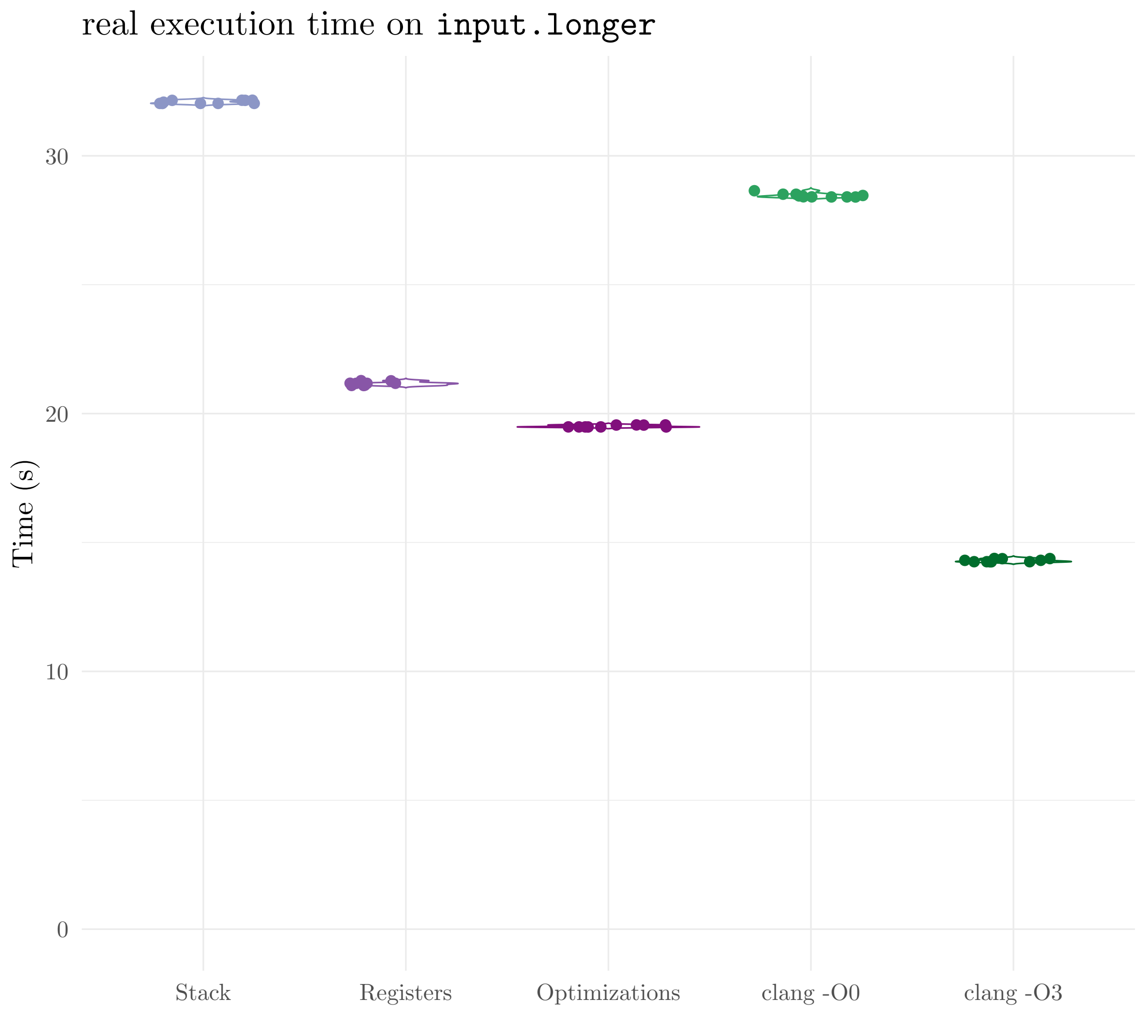

Figure 19: primes runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	32.03	32.03	32.06	32.09	32.15	32.16
Registers	21.09	21.11	21.17	21.17	21.18	21.28
Optimizations	19.48	19.48	19.48	19.51	19.56	19.56
clang -O0	28.40	28.41	28.43	28.46	28.50	28.65
clang -O3	14.25	14.25	14.28	14.30	14.36	14.38

Table 34: primes runtime (in seconds) summary statistic based on 10 runs each

4.18 programBreaker

	LLVM IR	ARM Assembly
Stack	84	137
Registers	41	107
Optimizations	36	100

Table 35: programBreaker instruction counts for generated code excluding labels

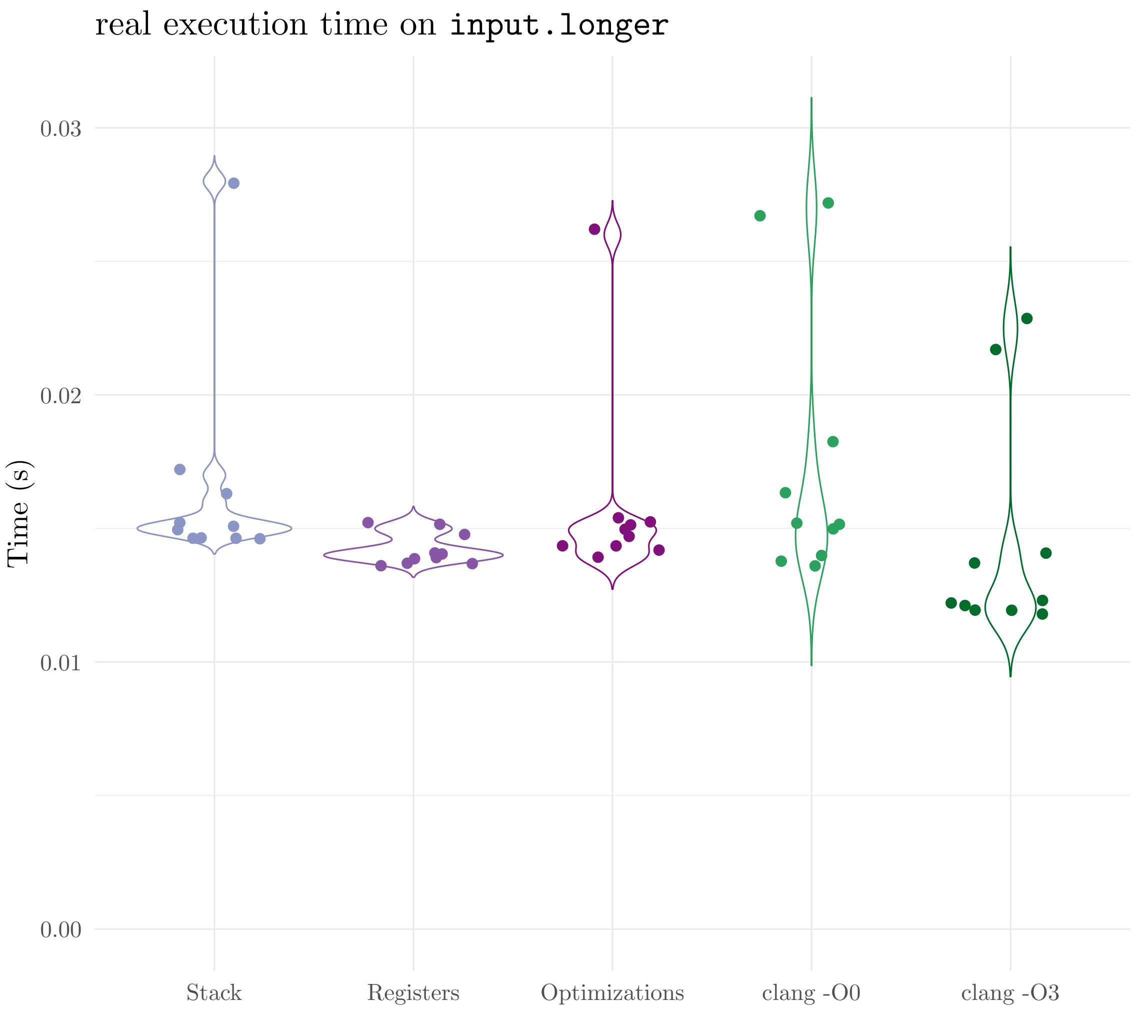

Figure 20: programBreaker runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.01	0.01	0.01	0.02	0.02	0.03
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.02	0.01	0.03
clang -O0	0.01	0.01	0.01	0.02	0.02	0.03
clang -O3	0.01	0.01	0.01	0.01	0.01	0.02

Table 36: programBreaker runtime (in seconds) summary statistic based on 10 runs each

4.19 stats

	LLVM IR	ARM Assembly
Stack	253	348
Registers	148	302
Optimizations	130	247

Table 37: stats instruction counts for generated code excluding labels

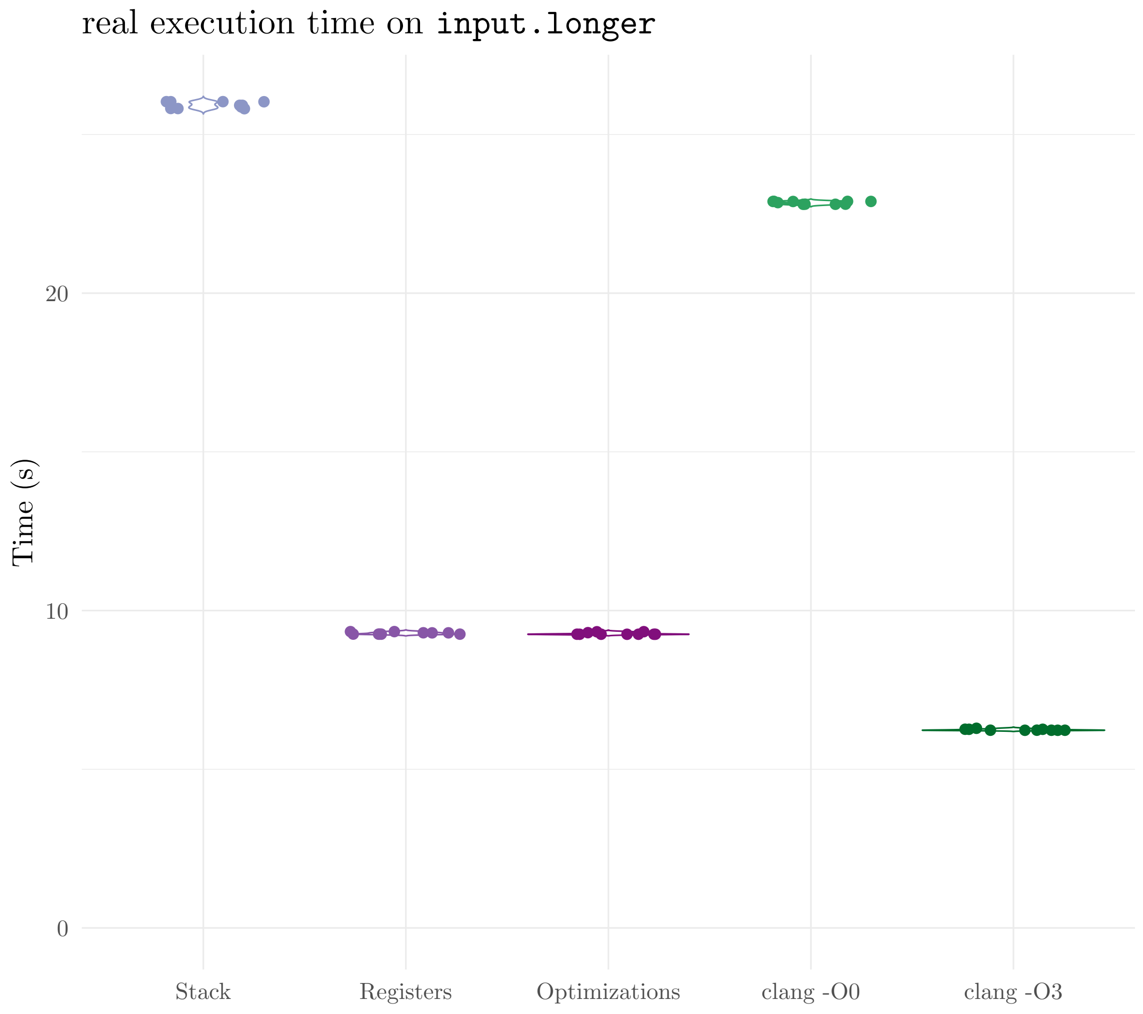

Figure 21: stats runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	25.81	25.83	25.92	25.93	26.03	26.03
Registers	9.26	9.26	9.28	9.29	9.30	9.34
Optimizations	9.25	9.25	9.25	9.27	9.29	9.33
clang -O0	22.80	22.80	22.87	22.85	22.89	22.89
clang -O3	6.23	6.23	6.23	6.24	6.26	6.29

Table 38: stats runtime (in seconds) summary statistic based on 10 runs each

4.20 wasteOfCycles

	LLVM IR	ARM Assembly
Stack	49	80
Registers	27	65
Optimizations	25	63

Table 39: wasteOfCycles instruction counts for generated code excluding labels

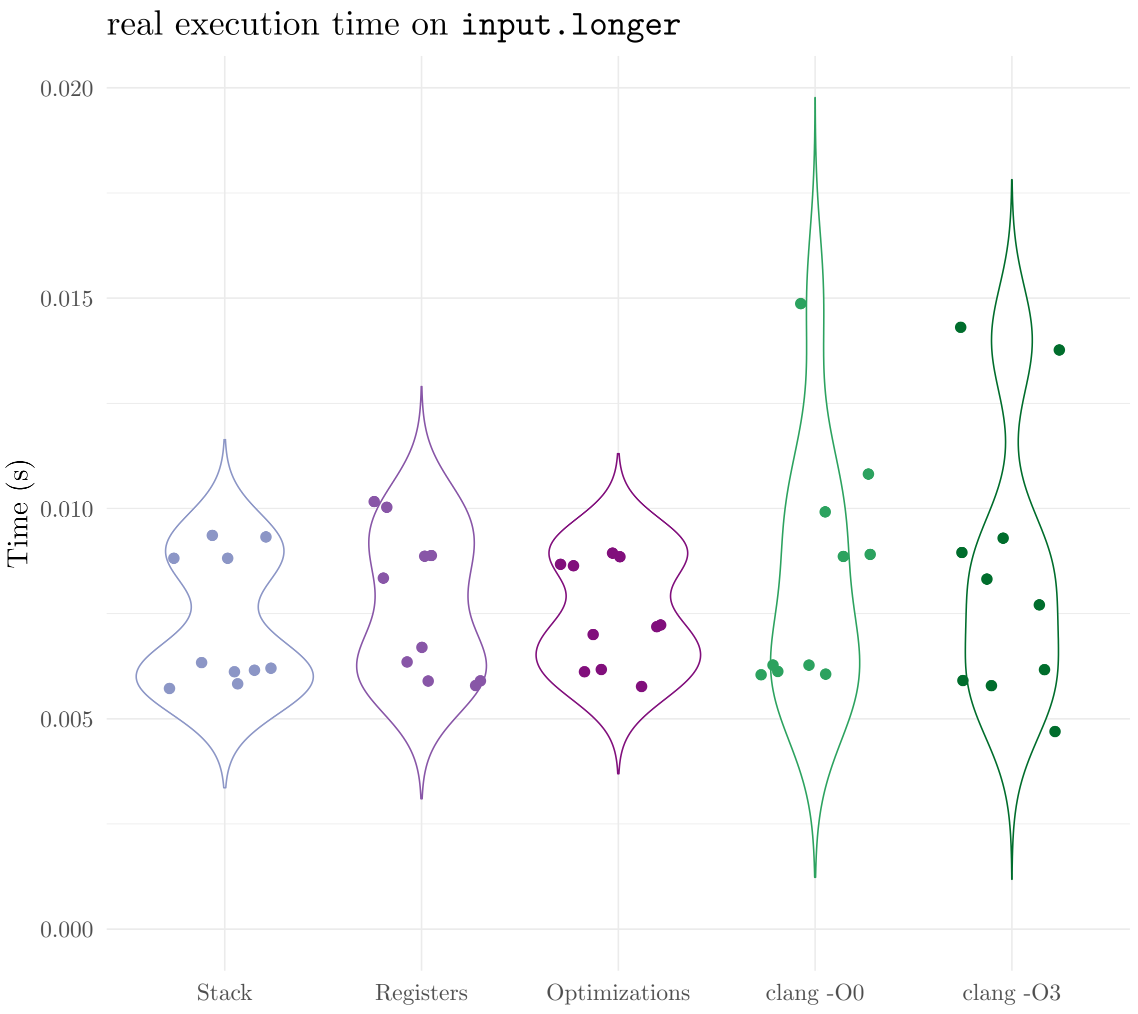

Figure 22: wasteOfCycles runtimes measured on 32bit ARM Raspberry Pis

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Stack	0.01	0.01	0.01	0.01	0.01	0.01
Registers	0.01	0.01	0.01	0.01	0.01	0.01
Optimizations	0.01	0.01	0.01	0.01	0.01	0.01
clang -O0	0.01	0.01	0.01	0.01	0.01	0.01
clang -O3	0.00	0.01	0.01	0.01	0.01	0.01

Table 40: wasteOfCycles runtime (in seconds) summary statistic based on 10 runs each