`mc`: A Mini Language Compiler in Go

Ellis Ruckman Jaxon Haws

June 17, 2023

Contents

1 Introduction

`mc` is a simple compiler for the Mini language, written in about 8,000 lines of Go. In most respects, Mini is a simplified version of C built specifically for the Compiler Construction course at Cal Poly. It features 64-bit integers, booleans, and structs, but lacks other important hallmarks of C like characters or arrays. Structs are always used as references, and memory for a struct can be allocated and freed with the `new` and `delete` keywords, respectively. Printing and reading integers from `stdin/stdout` is handled with the `read` and `print` constructs, which internally generate calls to `printf` and `scanf`.

The compiler features targets for both LLVM and 64-bit ARMv8. Additionally, the intermediate representation phase can be configured to use either stack-based IR or register-based IR depending on a command-line argument, and a variety of optimizations can be enabled or disabled at will.

2 Architecture

In terms of code, `mc` is split into roughly five sections: a parser package, an AST (abstract syntax tree) package, an IR (intermediate representation) package, an optimization package, and an ARM assembly package. All of these are combined in one Go module, as described by the top-level `go.mod` file. In Go, each source file belongs to one package, and all the files in a given package live in that package's namespace. Using multiple packages allowed us to avoid function and type naming collisions; for example, our code includes a different `Block` struct definition in both the `ir` and `asm` packages.

One of the main language features we used in our code was Go's interface type. In Go, an interface is a type that can hold any value (`struct`, `int`, `bool`, etc.) that implements a set of defined methods. For example, the `Stringer` interface type can hold any value that implements a particular `string` method. We used interfaces across our codebase to create pseudo-union types like `Instr` in the IR package, which can hold any other concrete instruction struct like `LoadInstr` or `CallInstr`. Go also includes built-in support for maps (hash tables) and slices (arrays), both of which we used extensively.

Since concurrency is such a major feature of Go, we decided to run pieces of our compiler in parallel in order to (hopefully) decrease compile times. Wherever a transformation or optimization is applied to multiple source functions, we create a goroutine to handle each one in parallel. This feature, along with the fact that iterating through a map in Go always returns items in an unstable order, means that our compiler creates subtly different output for each run. As you might guess, debugging in this environment was challenging at times, but we're proud to have a working multi-threaded compiler.

2.1 Parsing

The first phase of our compiler is the parser. Most of the legwork here is handled by ANTLR, a parser generator tool written in Java. With a single run, ANTLR can generate lexer and parser code for a given EBNF grammar in a multitude of languages. Luckily for us, a community member named Jim Idle maintains a Go target for ANTLR, which we utilized initially to generate parser code in Go. This portion of our code resides in the `mantlr` package (**M**ini **ANTLR**).

The generated parser is capable of transforming source code into a series of "context" structs and interfaces. This first form constitutes the parse tree. To create a simpler and cleaner surface for later transformations, the `parser` package contains code that translates the parse tree into a custom AST. The AST mainly consists of general interfaces (pseudo-union types) like `Statement` and `Expression` with concrete implementations like `PrintStatement` or `IntExpression`. Many of these structures contain pointers to other structures representing deeper portions of the AST, as in a `BinaryExpression` holding references to its left and right operand expressions.

As a nice bonus, translating from the parse tree to an AST allowed us to remove some knobby parts of the parse tree. All "nested expressions" (an expression surrounded by parentheses), for example, get flattened directly to expressions with proper hierarchy in the AST.

2.2 Static Semantics

Once the source has been translated to an AST, we leverage the tree to perform static type checking and return path checking. We first construct a symbol table containing mappings from identifiers to their declared types, as well as a struct table mapping struct names to their declared fields and types. We decided to include function types in the symbol table so that they can be checked later within the goroutine for each function body. Additionally, we used an ordered map package for the struct field mappings so that we could find the field index in the future for LLVM `getelementptr` instructions (Go's maps are not ordered by default).

2.2.1 Type Checking

Type checking for each function proceeds independently and begins with the creation of a local symbol table similar to the global version created above. We use Go's type switch construct

```
switch v := someInterface.(type) {
```

to dispatch a different function for each type of statement (`AssignmentStatement`, `InvocationStatement`, etc) encountered in a function body. These functions, in turn, call an expression handler function for each contained expression, and this function uses another type switch. It's switch statements all the way down!

When this recursive process returns upward, we return the type of each expression/lvalue and compare it against the expected type, generating and coalescing an error message for each mismatch encountered. Most statements expect a specific type for their arguments, as in `PrintStatement` expecting and integer or `AssignmentStatement` expecting matched types for both its lvalue and right expression. In cases where a type mismatch occurs, we return a special `ErrorType` instead of a "real" type like integer or boolean. `ErrorType` can convert to any of the other concrete types, which helps us avoid spewing extraneous error messages that would only confuse the user. In terms of effect, this means that an invalid expression like `(1 + true) + 5` only outputs a single error message for the inner type mismatch.

2.2.2 Return Checking

Return path checking works at the statement level. For a non-void function to be valid, at least one statement in its body must be return equivalent.

To help check for this condition, the main `typeCheckStatement` function that handles each statement passes back a "return equivalent" boolean flag. The functions for processing most concrete statements never set this flag, but some like `typeCheckReturnStatement` obviously do. For a more interesting case, `typeCheckBlockStatement` returns a true flag if one of its subsequent body statements is recursively return equivalent. Likewise, `typeCheckIfStatement` returns a true flag if both its "then" and "else" sections are recursively return equivalent.

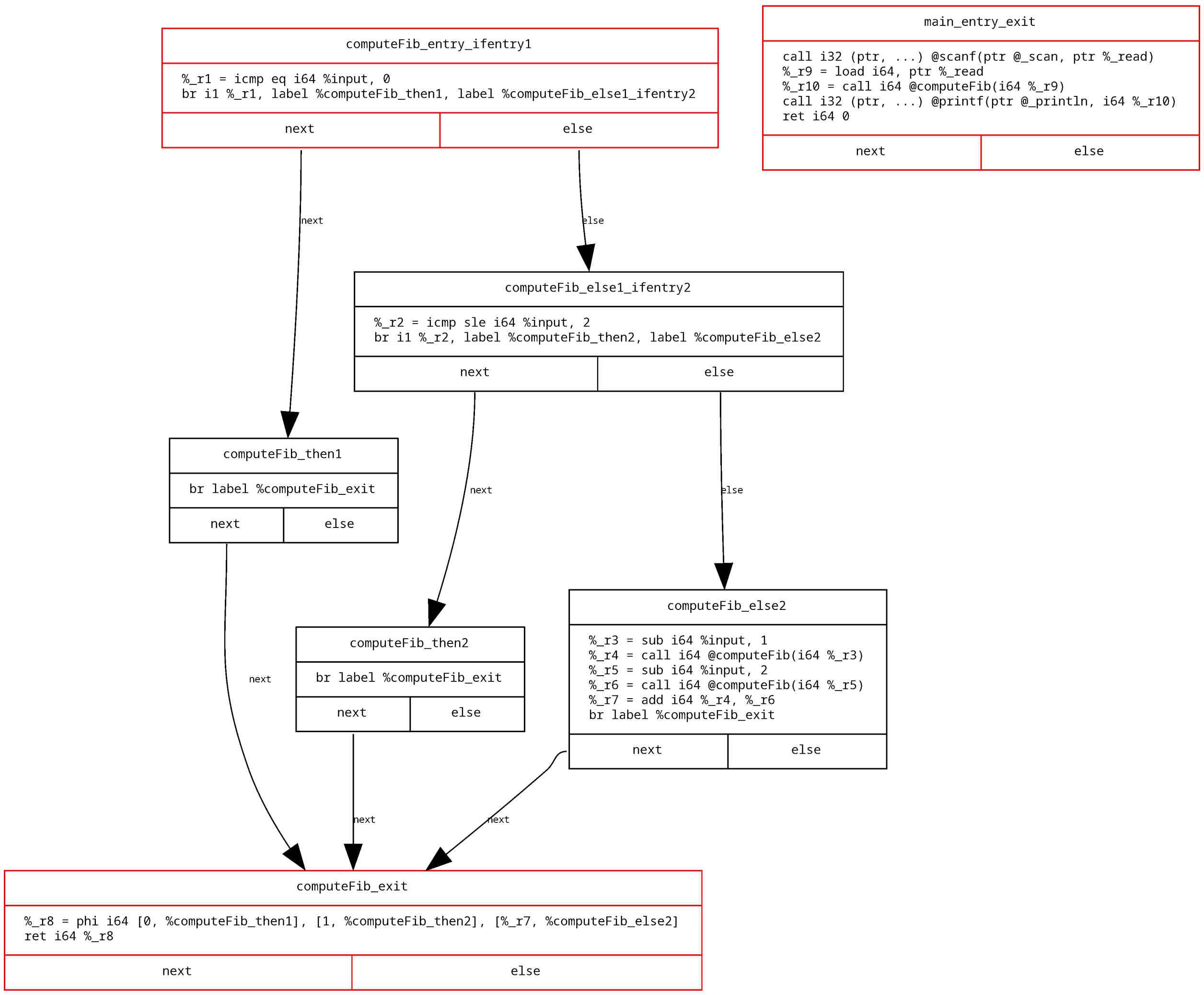
Here, we leverage Go's concept of *default values*. The declared but unset "return equivalent" flag is always false until it is set by one of the three functions above. Likewise, an unset integer in Go is aways 0 and an unset pointer is always `nil`.

2.3 Intermediate Representation

The next transformation pass creates an intermediate representation from the AST described above. The IR consists of a map of structs, a map of globals, and a map of functions. We opted to simply derive the first two from their equivalent symbol table and struct table maps created during the type checking phase of the compiler.

2.3.1 Control Flow Graphs

Each function contains a control flow graph (CFG), which is a pointer-linked graph of basic blocks. Basic blocks represent straight line code between each `if`, `while`, or `return` statement. As you can see in figure 1 below, each CFG has an "entry" block and an "exit" block (both marked in red).



```
computeFib_entry_ifentry1
%_r1 = icmp eq i64 %input, 0
br i1 %_r1, label %computeFib_then1, label %computeFib_else1_ifentry2

main_entry_exit
call i32 (ptr, ...) @scanf(ptr @_scan, ptr %_read)
%_r9 = load i64, ptr %_read
%_r10 = call i64 @computeFib(i64 %_r9)
call i32 (ptr, ...) @printf(ptr @_println, i64 %_r10)
ret i64 0

computeFib_else1_ifentry2
%_r2 = icmp sle i64 %input, 2
br i1 %_r2, label %computeFib_then2, label %computeFib_else2

computeFib_then1
br label %computeFib_exit

computeFib_else2
%_r3 = sub i64 %input, 1
%_r4 = call i64 @computeFib(i64 %_r3)
%_r5 = sub i64 %input, 2
%_r6 = call i64 @computeFib(i64 %_r5)
%_r7 = add i64 %_r4, %_r6
br label %computeFib_exit

computeFib_then2
br label %computeFib_exit

computeFib_exit
%_r8 = phi i64 [0, %computeFib_then1], [1, %computeFib_then2], [%_r7, %computeFib_else2]
ret i64 %_r8
```

Figure 1: Control flow graphs for the Fibonacci benchmark

We spent extra time on this portion of the program to make sure the CFG block labels are simple to read, as we knew we'd be looking at them throughout the rest of the milestones. Each label is generated dynamically from a slice (array) of strings representing the purpose of the block. An integer suffix on each stringensures the labels are globally unique and helps show which blocks are related.

Our compiler handles CFG generation in the `ir.processStatements` function, which loops through the statements in an AST function body. When the function encounters an `IfStatement` or `WhileStatement`, it creates new blocks for the "then" and "else" sections or the "while body" section and recursively calls itself to handle the inner AST statements.

We also tried to optimize this code to avoid creating unnecessary blocks. Our compiler handles `if` statements without an "else" section so that their `ifentry` block has a link straight to the corresponding `ifexit` (instead of through a trivial block). We also wrote code to combine exit blocks with their predecessor if they only have one previous block. You can see the result of this process in figure 1 above, where the `main` function has been reduced to a single block performing the role of "entry" and "exit."

To help us visualize our CFGs, we used the gographviz package to optionally output a description of each CFG in the dot language. dot can be parsed with a variety of programs to create graphs like the one shown above. Since we store the CFG for each function as a simple pointer to its entry block, this dot generation code performs a depth-first search through each CFG. We use this technique many times throughout the compiler to traverse tree and graph-like structures.

2.3.2 LLVM

Each CFG block contains a slice of instructions in an intermediate representation. For this project, we used LLVM as our IR target, since it is widely used in the industry and provides a well-rounded set of generic assembly constructs. Additionally, LLVM is much simpler than true assembly; thus, performing the initial translation from the AST to LLVM is comparatively easy (as opposed to a straight AST to ARM assembly translation). With an LLVM intermediate stage, writing translations for new assembly targets is also much more feasible; we could add an x64 target for `mc` with *relative* ease at this point.

In terms of implementation, our compiler creates LLVM instruction slices for each basic block at the same time as it runs CFG creation. `mc` translates non-special AST statements directly into their LLVM counterparts and uses calls to external C libraries to satisfy the rest. Expressions like an addition expression are translated to `BinaryInstr` with the appropriate operator set.

Each LLVM instruction is stored internally as a struct implementing the `ir.Instr` interface. In order to facilitate printing LLVM, each instruction also implements the `Stringer` interface and thus can be serialized to a string form. For example, a multiplication instruction for `x * 2` might be serialized as such:

```
bin := &BinaryInstr{
    Target:   &Register{
        Name: "_r1",
        Type: &IntType{64},
    },
    Operator: OrOperator,
    Op1:      &Register{
        Name: "x",
        Type: &IntType{64},
    },
    Op2:      &Literal{
        Value: "2",
        Type: &IntType{64},
    }
}

fmt.Printf("%v", bin)
// Output: %_r1 = mul i64 %x, 2
```

As you can see, the operands for most instructions are either `Registers` or `Literals`. The type of each value within the program is also encoded in these registers and literals. Unlike true assembly which must work with the finite number of registers offered by a CPU, LLVM assembly uses an infinite pool of *virtual registers*.

2.3.3 Single Static Assignment

One constraint imposed by LLVM is that no register can be assigned to more than once. This property is called "single static assignment," or SSA. When using an IR in SSA form, it is well-known where a register was first set; consequently, reading SSA code is quite easy. Contrast this state to the chaos of full register-allocated assembly: general registers are re-used many times and determining the path of a particular piece of data is quite difficult. The comparative ease of reading SSA also applies in practice during the compiler optimization phase, where an IR in SSA form generally presents more ooportunities for optimizations than a non-SSA counterpart. See section 2.4 for more evidence of this fact.

To fully utilize the benefits of LLVM's forced SSA format, our register structs contain a single pointer to their defining instruction and a slice of pointers to instructions where the register is used. This web of "defs" and "uses" forms what we call the *SSA overlay graph*. To help us visualize register defs and uses, we added an optional flag to `mc` that instructs it to print out the def-use chains for each register after translating to LLVM.

Our compiler supports two methods for ensuring that the generated LLVM is in SSA form. The first is to store every local variable on the stack, and simply load and store for each would-be register access. LLVM does not include memory as part of its SSA requirements, so variables placed on the stack are free from scrutiny. This method is simple but reduces the effective scope of optimizations and forces the resulting program to perform many costly memory accesses.

The second option is to keep a map of known variables in each CFG block and insert phi (ϕ) instructions to marshall movement of known variables from predecessor blocks to their successors. Our compiler has separate `assignmentStatement-ToLlvmStack` and `assignmentStatementToLlvm-Reg` functions, along with separate functions for identifier expressions, function initialization, and function completion. Phi instructions get inserted when looking up an unknown identifier in a block with two or more predecessors, as each different predecessor block might contain a different value for that variable.

2.4 Optimization

`mc` has three optimizations enabled by default, each of which operates on and simplifies the LLVM intermediate representation.

2.4.1 Constant Propagation

Constant propagation is an optimization designed to remove simple clusters of constant expressions like `"(1 + 2) * 3"` and replace them with an equivalent constant. The SSA overlay graph came in very handy for this optimization; replacing a register with an evaluated literal is as simple as replacing all of the register's uses with the literal.

For `mc`, we decided to implement a specific constant propagation algorithm called "sparse conditional" constant propagation. This algorithm not only folds and propagates constants to the locations where they are used, but also incidentally analyzes the CFG to find blocks which will never run. The compiler uses this block-level information to unlink non-executable blocks and remove phi values which come from those blocks.

In order to fold constant expressions, we implemented a very basic set of evaluation functions as methods on the `Literal` struct. The methods `doCondition` and `doOperation` handle the bulk of this work, alongside smaller additions like `toBool`. These methods let us act as an interpreter to evaluate LLVM arithmetic at compile-time.

2.4.2 Useless Code Elimination

The purpose of useless code elimination is to detect and remove code that serves no purpose. The algorithm for this optimization follows a mark and sweep pattern. In the mark phase, critical instructions like `RetInstr`, `CallInstr`, and `StoreInstr` are marked as "useful." Subsequently, any instructions which act as the defintion for registers in these critical instruction are marked as well, and the process continues recursively until all "useful" instructions are marked. In the sweep phase, any non-marked instructions are removed, as they will never affect the visible/detectable output of the program.

2.4.3 Trivial Phi Removal

Phi instructions are used to resolve conflicts between predecessor blocks, where one block thinks a register should be "x" and the other believes it should be "y". A phi instruction is trivial if all its possible values are equivalent, since the exact phi output can be found at compile time.

Trivial phi removal, logically, is a simple but effective optimization designed to remove these trivial phis from an SSA IR. Once a trivial phi is removed, its target can be rewritten to use the known value instead. Since this re-write operation may expose new trivial phis, the process is run in a loop until no further changes are required.

2.5 Assembly Generation

The final transformation phase in our compiler is the translation from high-level LLVM assembly to ARMv8 assembly. As you might expect by now, the final ARM form is represented in `mc` by a set of instruction structs that implement the `asm.Instr` interface. Each instruction contains pointers to operands which are either `Registers` or `Immediates` (registers in the `asm` package are separate from those in the `ir` package). Each register is marked as either virtual or physical. In the first stage of assembly generation, virtual registers are used alongside physical registers.

2.5.1 Stack Management

One of the trickiest parts of dealing with true assembly is managing the stack. Each assembly function begins with a prologue containing an instruction to store the previous frame pointer and linker register as well as an instruction to move the frame pointer to the current stack pointer:

```
    stp fp, lr, [sp, -16]!
    mov fp, sp
```

The full stack frame for a function call ends up with the following sections:

```
+------------------------+
| extra stack params     |
| ...                    |
+------------------------+
| saved lr               |
+------------------------+
| saved fp               |
+------------------------+ <- fp
| callee-saved regs      |
| ...                    |
+------------------------+
| spilled regs           |
| ...                    |
+------------------------+
| local stack vars       |
| ...                    |
+------------------------+ <- sp
```

The load/store addresses for memory instructions in ARMv8 are calculated by adding the value in a base register with an immediate offset. After the prologue above, every function pushes its callee-saved registers to the stack, using the frame pointer as the base and a pre-computable negative immediate for the offset. We specifically placed the callee-saved registers below the frame pointer so their addresses would be easy to compute and so none of the loads/stores for saving callee-saved registers would use an offset below -256 (this is, for the most part, not allowed in ARMv8). At this point in the compiler, all values are treated as 64-bits wide.

The offsets for local stack variables, meanwhile, are easily computed since they are directly above the stack pointer. Positive immediate offsets in ARMv8 can range up to 4096 (with asterisks); thus, keeping locals variables above a base register would hypothetically allow a compiled program to utilize a larger set of local stack variables. Whether you *should* have more than 32 local variables (again, the maximum negative offset allowed is -256) is open for interpretation.

Our compiler does not support spilled registers, but—if they were handled—they would be saved above the local stack variables. This position in the frame works well for spills because the maximum local stack variable offset is known by the time spills are created. Thus, spilled register offsets could be calculated based on this maximum local offset.

2.5.2 ARM Translation

When translating from LLVM to ARM, our compiler stores a mapping of most LLVM register names to their corresponding virtual ARM register. Any LLVM registers *not* in the map are instead in the `stackVars` table, which maps an LLVM register name to a base register and immediate offset used for stores and loads. This second table holds information for all the variables which must (unfortunately) be stored on the stack, including spilled registers and arguments for functions with more than eight parameters.

Much of the complicated code in our assembly generation phase deals with loading and storing to memory. In that regard, `mc` includes several functions that attempt to either move a register to a destination, or—if the register in fact resides on the stack—loads the value instead. If we did include support for spills, these functions would be ready to use for that case.

Since phi instructions do not exist in actual ISAs, the compiler must replace them with an equivalent construct during assembly generation. In `mc`, phi instructions in each CFG block are replaced by a `MovInstr` from a temporary register. Another `MovInstr` or `LoadInstr` is placed near the end of each previous ARM block to load this temporary register with an appropriate value, simulating the action of a phi.

When processing an LLVM instruction that will become an ARM `ArithInstr` (`add`, `sub`, `mul`, etc) or a `CompInstr` (`cmp`), the compiler makes an effort to optimize immediate operands. Each of these instructions (with a variety of maddening edge cases) can utilize a 12-bit immediate as its second operand. An addition instruction with an immediate might look like the follwing:

```
add _r1, _r2, 42
```

If the immediate does not fit in 12 bits, the compiler will attempt to use a move instruction with a 16-bit immediate instead:

```
mov _tmp1, 700
add _r1, _r2, _tmp2
```

Finally, if 16-bits is still not enough, the compiler will output a `LoadImmediate` pseudo-instruction which loads the 64-bit value from memory:

```
ldr _tmp1, =60000
add _r1, _r2, _tmp2
```

The compiler also attempts to use the zero register when possible and will swap operands for commutative operations to place an immediate into the second slot.

2.5.3 Register Allocation

To complete ARM translation, the compiler must map and rename virtual registers to appropriate general ARM registers (`x0-x28`). Since each ARM instruction struct implements the `asm.Instr` interface, they each have methods which return their target and source registers. Our register allocation algorithm uses this target and source information to easily create `genSet`s and `removeSets` for each ARM block. The compiler then computes the live range of each register (in the form of liveouts for each block) using these two sets in an iterative dataflow analysis process.

The interference graph for a function consists of a map of registers to `node` structs. A node contains a map of neighboring nodes and a set containing the colors already used by said neighbors. If a node is colored, it actually goes to each of its neighbors and adds its new color to their set of known neighbor colors. This speeds up coloring using the power of O(1) membership checks.

The compiler removes nodes from the graph based on their degree and pushes the virtual ones into a stack. Physical nodes are pre-colored, and then each virtual node is popped from the stack and colored, i.e. mapped to its final physical register.

3 Analysis

An important factor to consider when comparing compilers is the performance of the code they generate. To test `mc`, we compiled 20 different

Mini benchmark source files with a variety of configurations. To produce runnable binaries, we took the assembly files generated by `mc` and used an `aarch64-gcc` cross-compiler (GCC version 12.2.1) to assemble and link the final binaries. We used the `-static` and `-no-pie` flags when calling `gcc` for this final stage.

To provide additional data for comparisons, we also compiled C versions of each of the 20 benchmarks using the same `gcc` cross-compiler with the same `-static` and `-no-pie` flags. The final set of configurations is show below:

- Mini via `mc` (stack-based, no optimizations)
- Mini via `mc` (register-based, no opt)
- Mini via `mc` (register-based, optimizations)
- C via `gcc` (`-O0`)
- C via `gcc` (`-O3`)

Benchmark tests using these compiled binaries were performed on the Cal Poly ARM UNIX server during a time in which no other users (besides root, the chron user, etc) were online. The ARM server is a 96-core machine with 64 gigabytes of RAM. For our testing, we used a `ksh93` shell script that launched 100 subprocesses to run the 100 binaries in parallel. Each subprocess timed the execution of its designated binary using the `ksh` idiom

```
typeset -F SECONDS=0; <cmd>; echo "$SECONDS"
```

as described by this StackOverflow post. Using `ksh` here allowed us to measure execution time with microsecond precision (note: precision, not accuracy).

To guard against fluke results, we ran our test script 10 times and collected the results for each run. See sections 3.2–3.21 below for information on the performance of different benchmarks in each of the five tested configurations.

3.1 Results

Overall, the benchmark timing results show that `mc` in many cases, even our un-optimized stack-based output has similar speed to a zero-optimization GCC binary. For the majority of benchmarks that ran for over one second, both the register-based (optimized and unoptimized) versions ran faster than their corresponding zero-optimization GCC binary. Of course, the `-O3` gcc binaries were always far and ahead the fastest out of the lot.

Enabling register-based code generation instead of stack-based in `mc` seemed to almost always have a large positive effect on generated binary runtime. The runtime effect of enabling optimizations was less pronounced, but is still visible in many of the longer graphs. In terms of code size, enabling register-based IR or optimizations always reduced the generated number of ARM instructions (even if just by a small margin). This effect is most pronounced in table 30 for OptimizationBenchmark.

3.2 BenchMarkishTopics

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	101.6552	1.7672
`mc`, register-based	45.2794	0.5036
`mc`, register-based, optimized	46.4307	0.9946
`gcc -O0`	66.5141	0.9601
`gcc -O3`	8.1762	0.0093

Table 1: Average runtimes for the BenchMarkishTopics benchmark

Program/Options	Instruction Count
`mc`, stack-based	191
`mc`, register-based	171
`mc`, register-based, optimized	163

Table 2: Assembly instruction counts for the BenchMarkishTopics benchmark

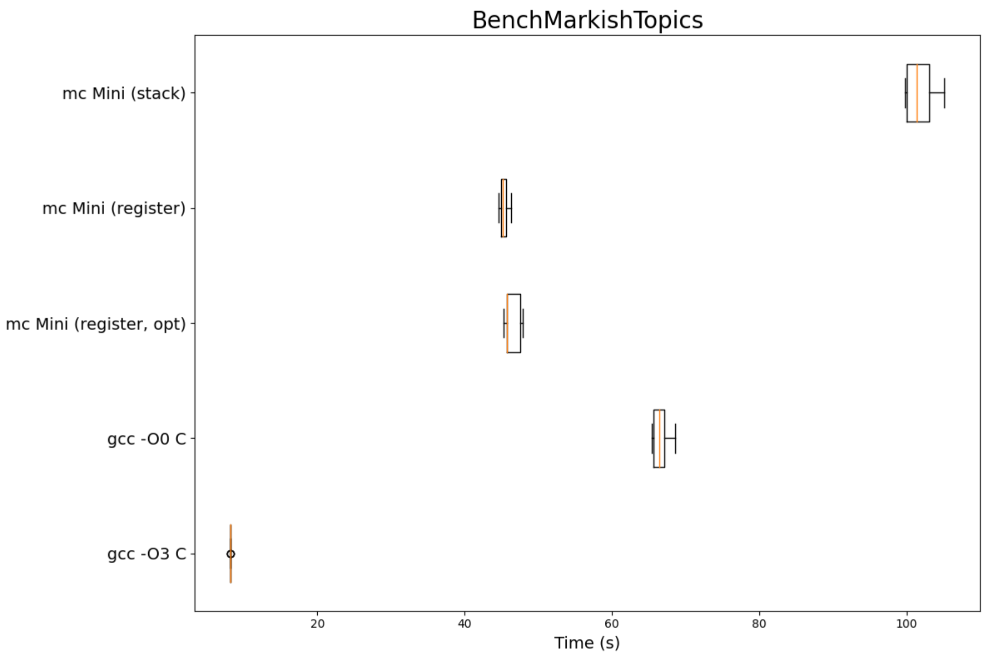


Figure 2: Runtimes for the BenchMarkishTopics benchmark (lower is better)

3.3 bert

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0154	0.0231
`mc`, register-based	0.0119	0.0119
`mc`, register-based, optimized	0.0119	0.0115
`gcc -O0`	0.0139	0.0156
`gcc -O3`	0.0121	0.0123

Table 3: Average runtimes for the bert benchmark

Program/Options	Instruction Count
`mc`, stack-based	1097
`mc`, register-based	978
`mc`, register-based, optimized	875

Table 4: Assembly instruction counts for the bert benchmark

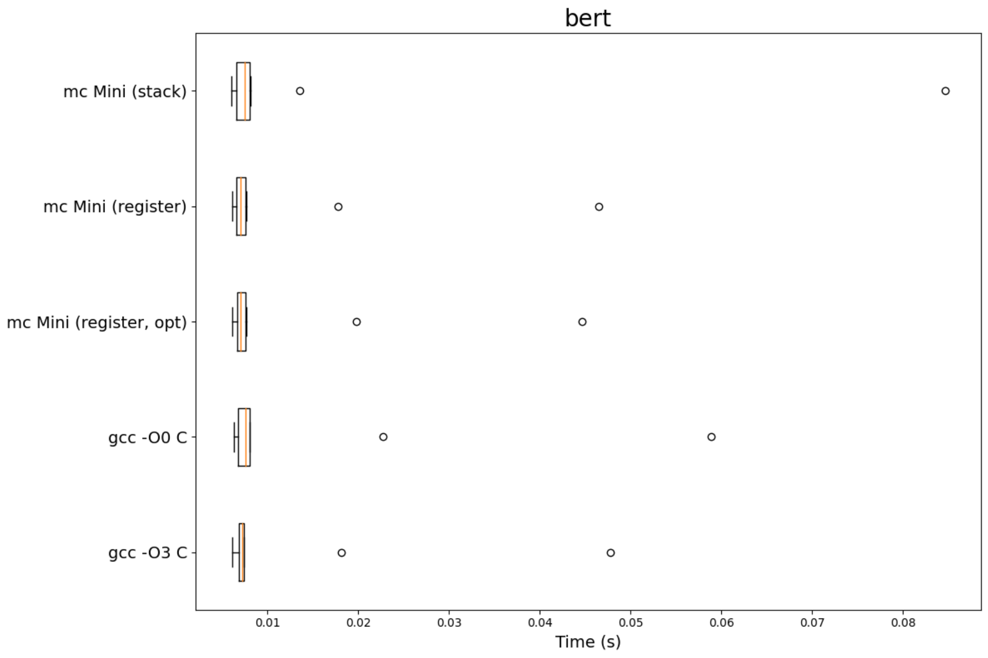


Figure 3: Runtimes for the bert benchmark (lower is better)

3.4 biggest

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0117	0.0092
`mc`, register-based	0.0116	0.0104
`mc`, register-based, optimized	0.0120	0.0103
`gcc -O0`	0.0117	0.0109
`gcc -O3`	0.0114	0.0100

Table 5: Average runtimes for the biggest benchmark

Program/Options	Instruction Count
`mc`, stack-based	127
`mc`, register-based	109
`mc`, register-based, optimized	100

Table 6: Assembly instruction counts for the biggest benchmark

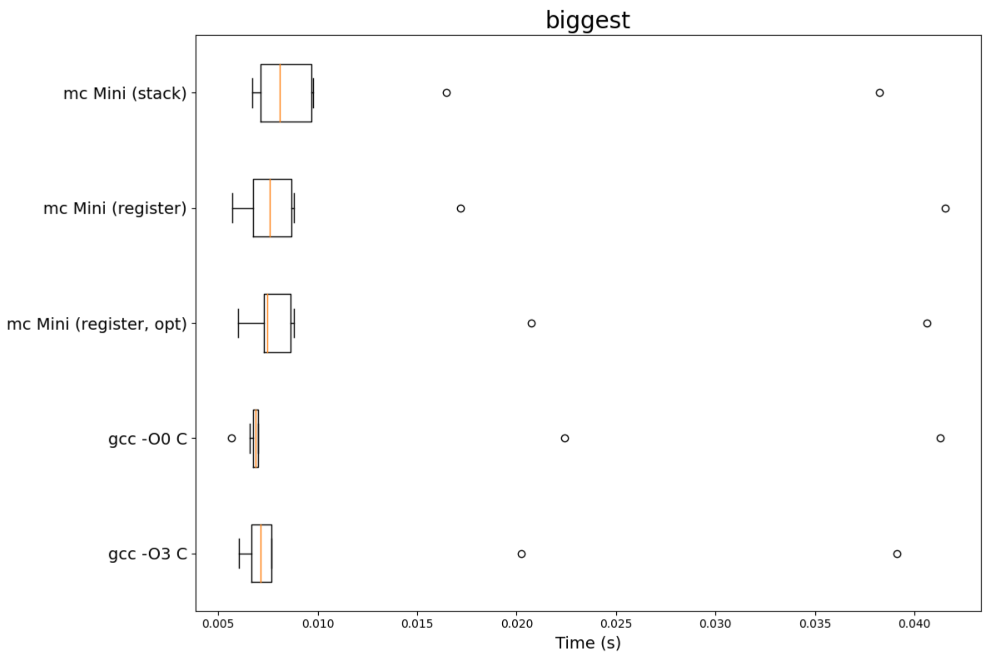


Figure 4: Runtimes for the biggest benchmark (lower is better)

3.5 binaryConverter

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	18.6099	0.0151
`mc`, register-based	10.6371	0.0055
`mc`, register-based, optimized	10.6390	0.0079
`gcc -O0`	18.6057	0.0051
`gcc -O3`	0.0074	0.0024

Table 7: Average runtimes for the binaryConverter benchmark

Program/Options	Instruction Count
`mc`, stack-based	193
`mc`, register-based	165
`mc`, register-based, optimized	146

Table 8: Assembly instruction counts for the binaryConverter benchmark

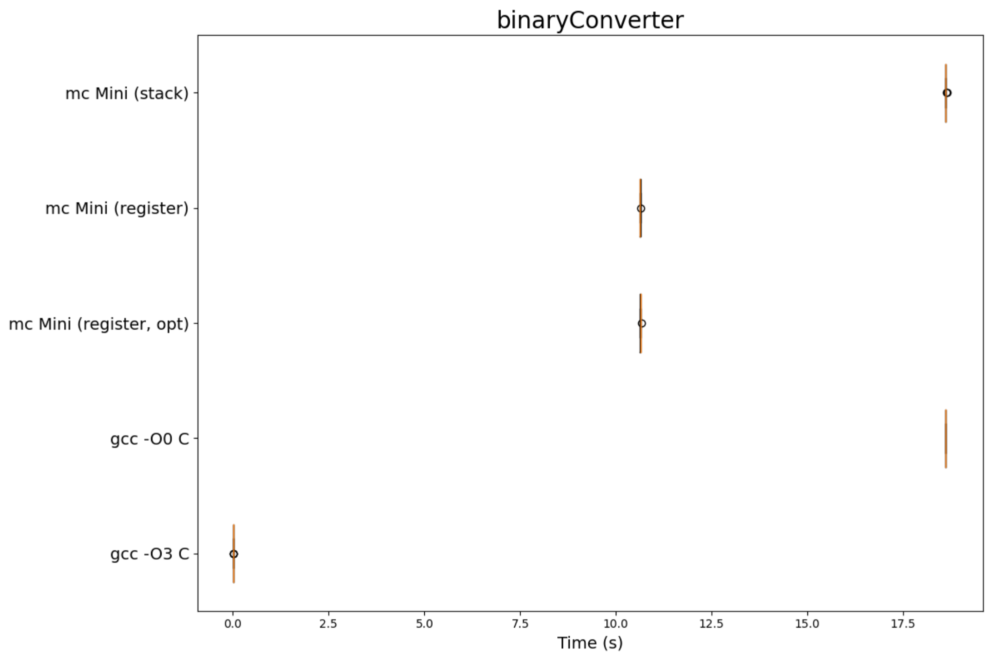


Figure 5: Runtimes for the binaryConverter benchmark (lower is better)

3.6 brett

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0078	0.0009
`mc`, register-based	0.0071	0.0008
`mc`, register-based, optimized	0.0073	0.0010
`gcc -O0`	0.0066	0.0006
`gcc -O3`	0.0069	0.0006

Table 9: Average runtimes for the brett benchmark

Program/Options	Instruction Count
`mc`, stack-based	1133
`mc`, register-based	1050
`mc`, register-based, optimized	717

Table 10: Assembly instruction counts for the brett benchmark

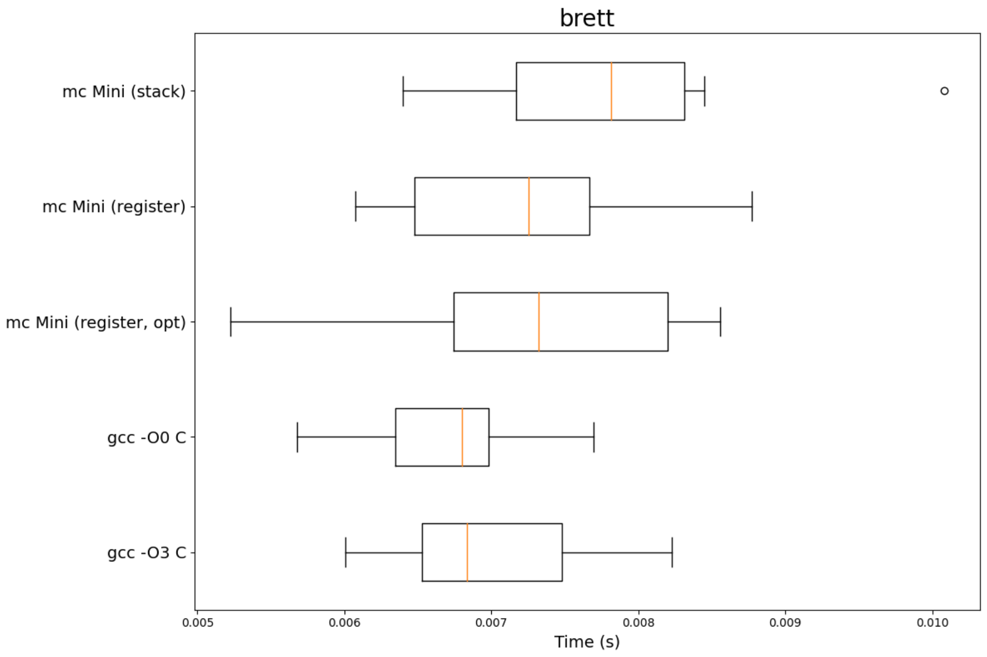


Figure 6: Runtimes for the brett benchmark (lower is better)

3.7 creativeBenchMarkName

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	5.1173	0.0036
`mc`, register-based	4.3879	0.0055
`mc`, register-based, optimized	2.9284	0.0035
`gcc -O0`	5.1215	0.0136
`gcc -O3`	0.0095	0.0040

Table 11: Average runtimes for the creativeBenchMarkName benchmark

Program/Options	Instruction Count
`mc`, stack-based	281
`mc`, register-based	250
`mc`, register-based, optimized	215

Table 12: Assembly instruction counts for the creativeBenchMarkName benchmark

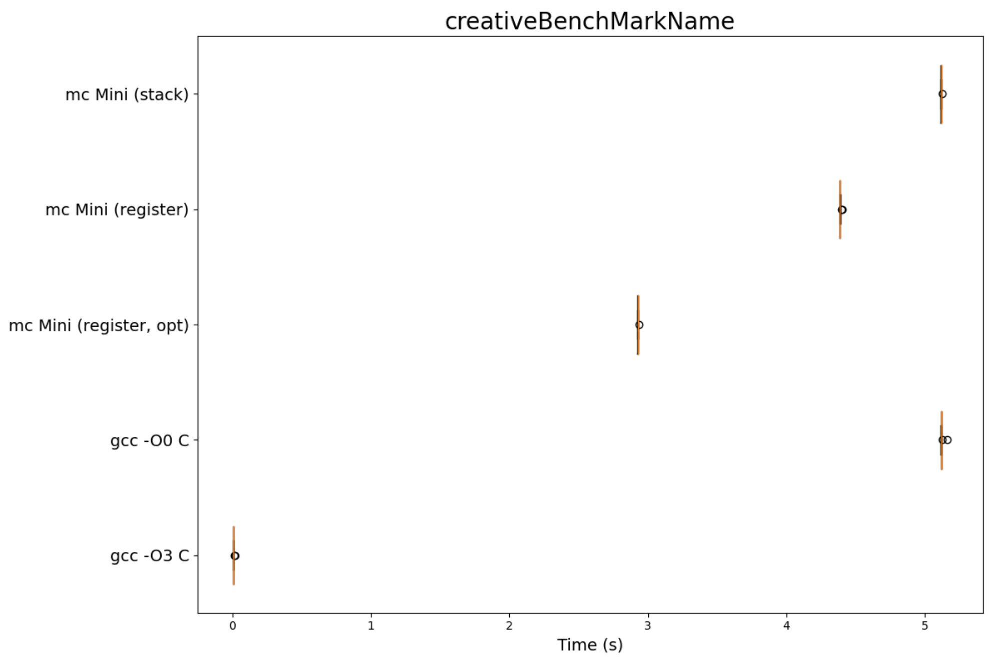


Figure 7: Runtimes for the creativeBenchMarkName benchmark (lower is better)

3.8 fact_sum

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0254	0.0072
`mc`, register-based	0.0099	0.0016
`mc`, register-based, optimized	0.0104	0.0016
`gcc -O0`	0.0097	0.0013
`gcc -O3`	0.0102	0.0011

Table 13: Average runtimes for the fact_sum benchmark

Program/Options	Instruction Count
`mc`, stack-based	121
`mc`, register-based	100
`mc`, register-based, optimized	92

Table 14: Assembly instruction counts for the fact_sum benchmark

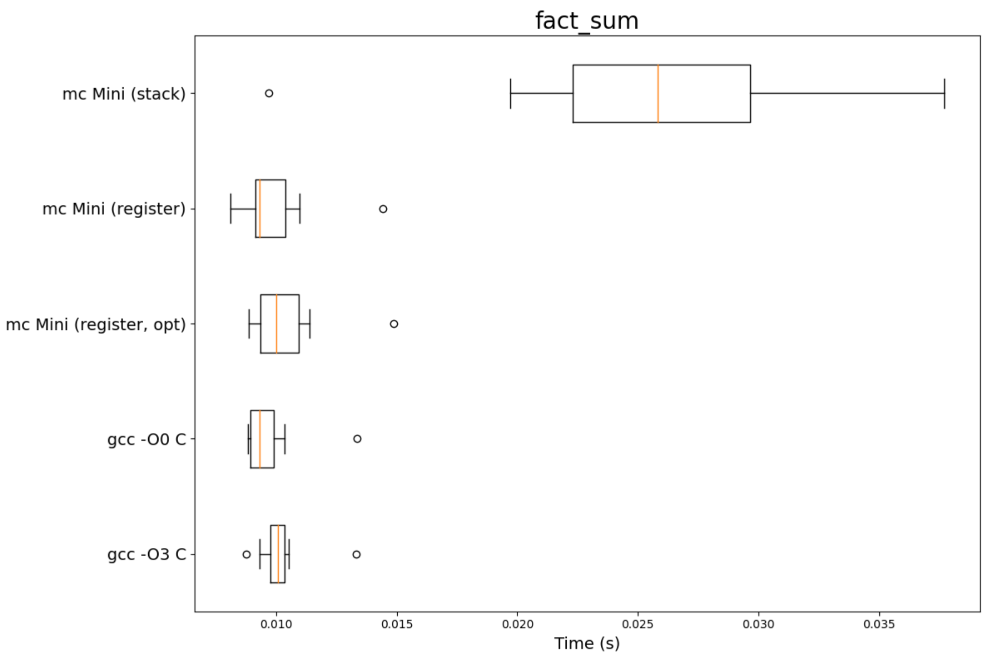


Figure 8: Runtimes for the fact_sum benchmark (lower is better)

3.9 Fibonacci

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	27.7501	0.0061
`mc`, register-based	25.1031	0.0283
`mc`, register-based, optimized	23.6642	0.0137
`gcc -O0`	21.6407	0.0071
`gcc -O3`	12.3436	0.0083

Table 15: Average runtimes for the Fibonacci benchmark

Program/Options	Instruction Count
`mc`, stack-based	61
`mc`, register-based	53
`mc`, register-based, optimized	51

Table 16: Assembly instruction counts for the Fibonacci benchmark

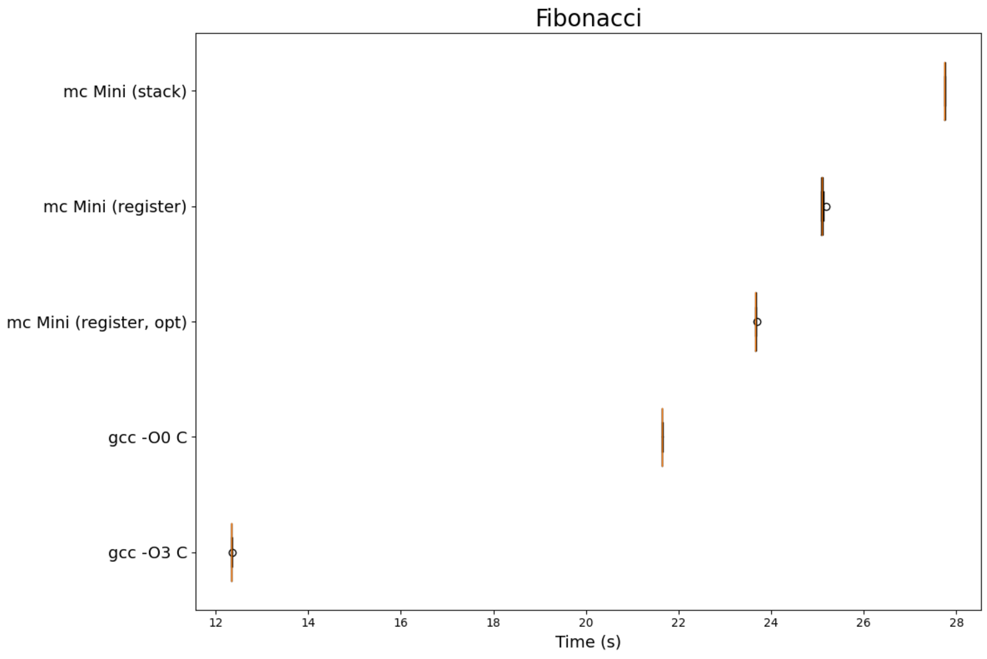


Figure 9: Runtimes for the Fibonacci benchmark (lower is better)

3.10 GeneralFunctAndOptimize

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	4.1868	0.0057
`mc`, register-based	2.7857	0.0259
`mc`, register-based, optimized	2.3356	0.0160
`gcc -O0`	2.0657	0.0120
`gcc -O3`	0.0617	0.0076

Table 17: Average runtimes for the GeneralFunctAndOptimize benchmark

Program/Options	Instruction Count
`mc`, stack-based	201
`mc`, register-based	208
`mc`, register-based, optimized	144

Table 18: Assembly instruction counts for the GeneralFunctAndOptimize benchmark

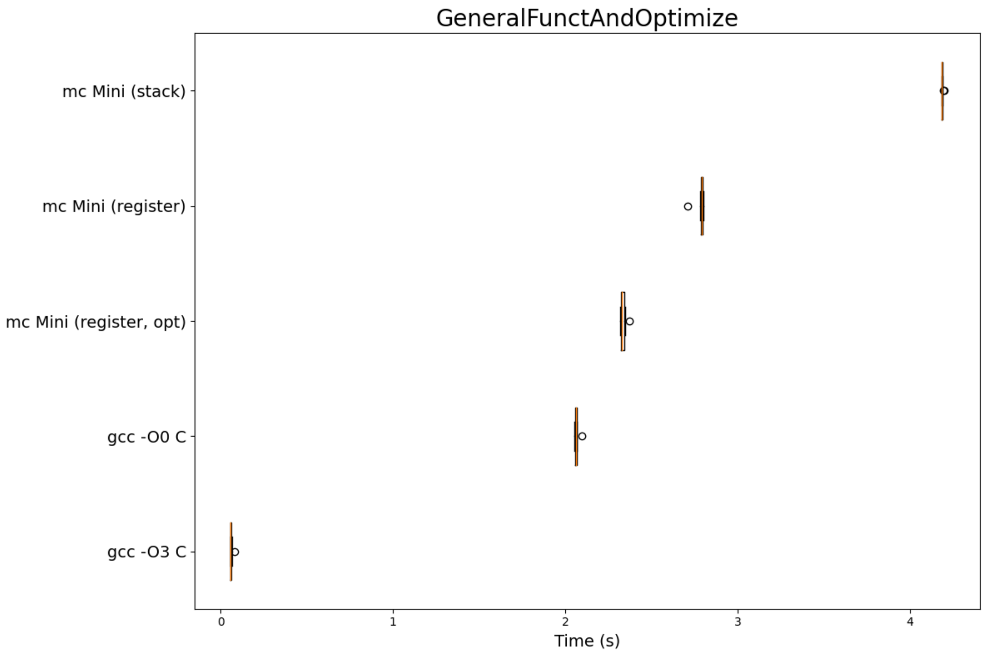


Figure 10: Runtimes for the GeneralFunctAndOptimize benchmark (lower is better)

3.11 hailstone

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0076	0.0014
`mc`, register-based	0.0072	0.0009
`mc`, register-based, optimized	0.0076	0.0008
`gcc -O0`	0.0071	0.0010
`gcc -O3`	0.0071	0.0007

Table 19: Average runtimes for the hailstone benchmark

Program/Options	Instruction Count
`mc`, stack-based	93
`mc`, register-based	79
`mc`, register-based, optimized	72

Table 20: Assembly instruction counts for the hailstone benchmark

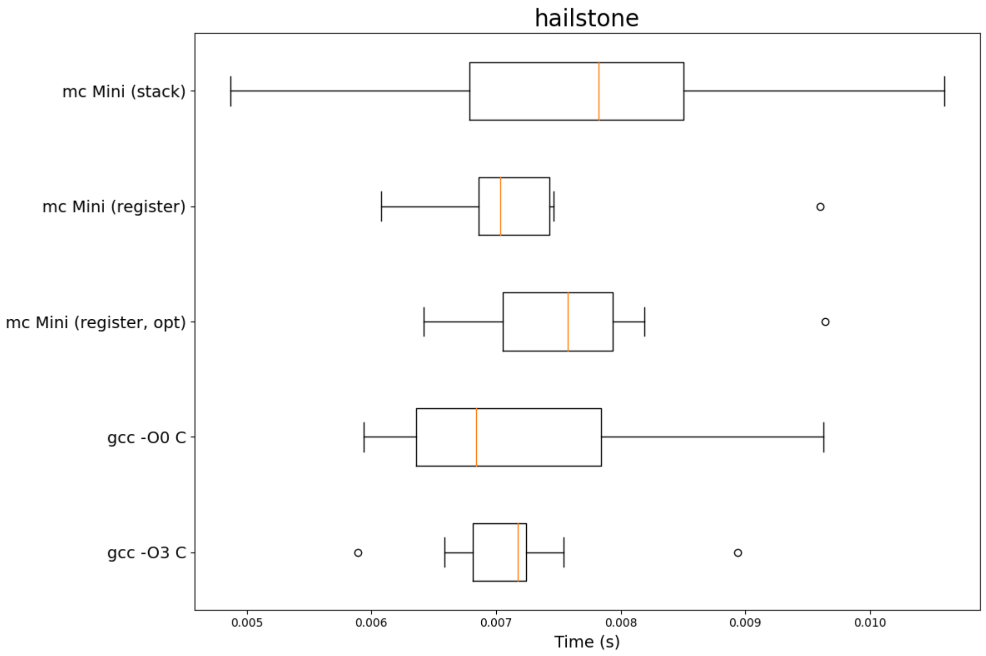


Figure 11: Runtimes for the hailstone benchmark (lower is better)

3.12 hanoi_benchmark

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	23.8539	0.0085
`mc`, register-based	19.7736	0.0129
`mc`, register-based, optimized	19.2505	0.0241
`gcc -O0`	17.9428	0.0057
`gcc -O3`	5.9484	0.0023

Table 21: Average runtimes for the hanoi_benchmark benchmark

Program/Options	Instruction Count
`mc`, stack-based	301
`mc`, register-based	274
`mc`, register-based, optimized	262

Table 22: Assembly instruction counts for the hanoi_benchmark benchmark

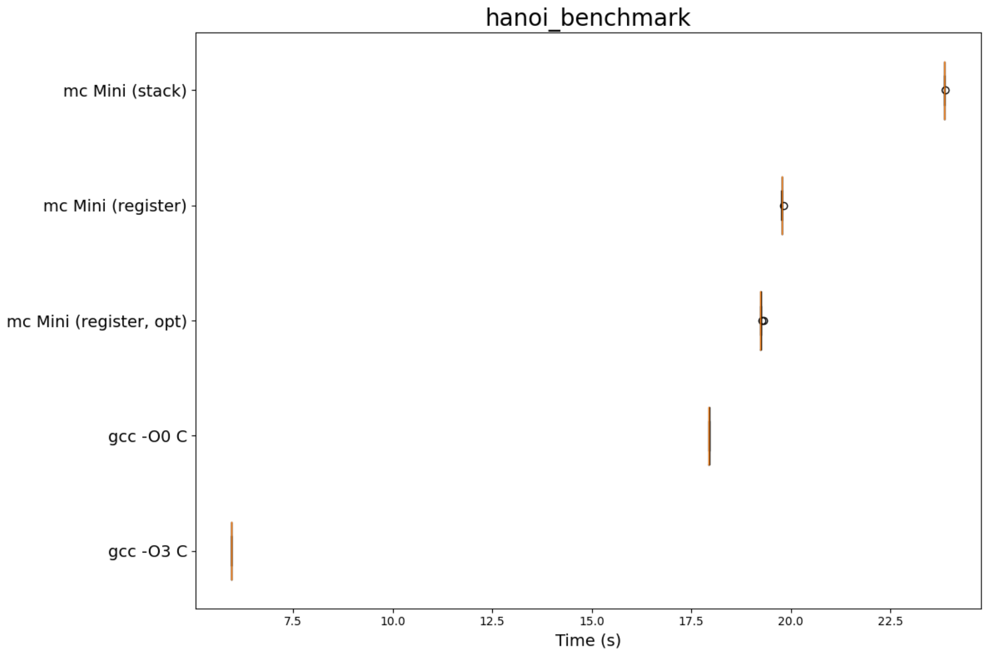


Figure 12: Runtimes for the hanoi_benchmark benchmark (lower is better)

3.13 killerBubbles

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	21.5402	0.2982
`mc`, register-based	14.3215	0.0629
`mc`, register-based, optimized	13.8393	0.0484
`gcc -O0`	17.6319	0.3543
`gcc -O3`	5.1261	0.0102

Table 23: Average runtimes for the killerBubbles benchmark

Program/Options	Instruction Count
`mc`, stack-based	219
`mc`, register-based	196
`mc`, register-based, optimized	163

Table 24: Assembly instruction counts for the killerBubbles benchmark

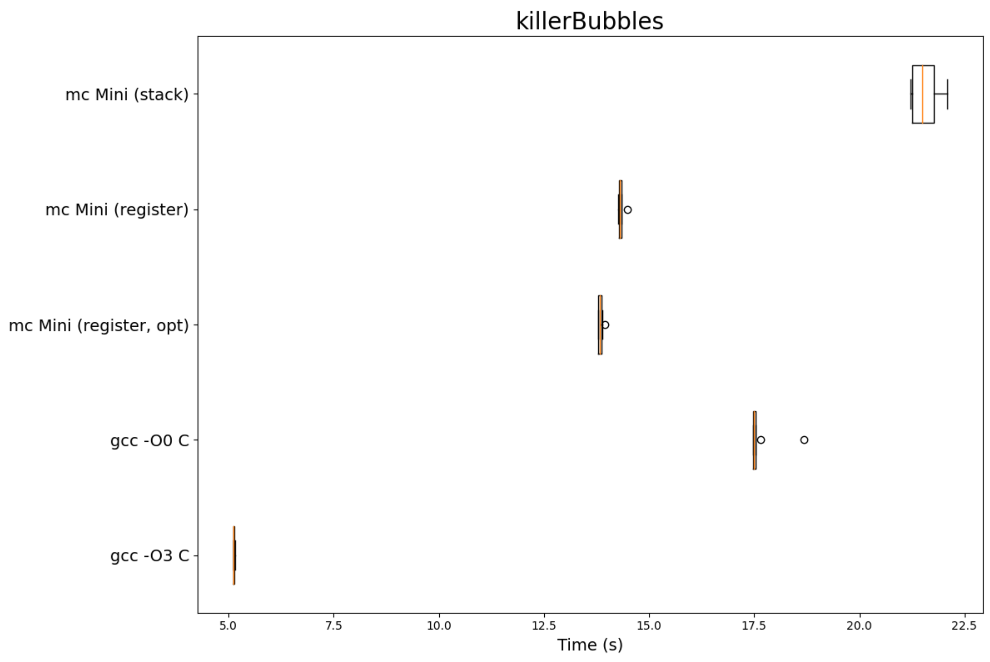


Figure 13: Runtimes for the killerBubbles benchmark (lower is better)

3.14 mile1

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	16.0492	0.0054
`mc`, register-based	7.0347	0.0042
`mc`, register-based, optimized	7.0343	0.0079
`gcc -O0`	16.0390	0.0038
`gcc -O3`	4.0164	0.0015

Table 25: Average runtimes for the mile1 benchmark

Program/Options	Instruction Count
`mc`, stack-based	100
`mc`, register-based	97
`mc`, register-based, optimized	80

Table 26: Assembly instruction counts for the mile1 benchmark

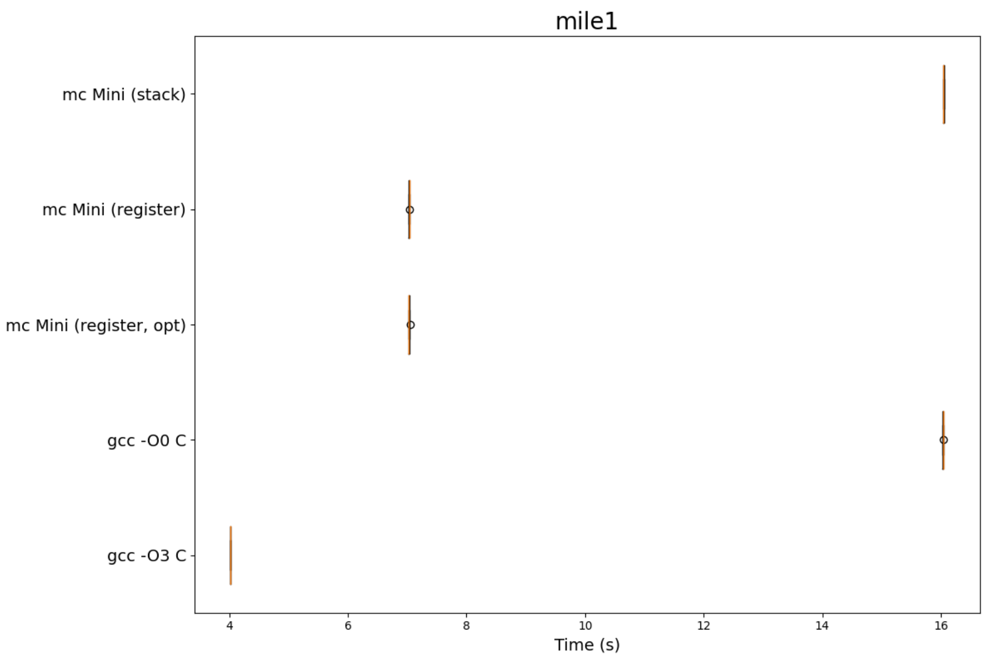


Figure 14: Runtimes for the mile1 benchmark (lower is better)

3.15 mixed

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	27.2677	1.5584
`mc`, register-based	13.9248	0.0257
`mc`, register-based, optimized	12.7299	0.0274
`gcc -O0`	20.0709	0.0241
`gcc -O3`	2.2406	0.0039

Table 27: Average runtimes for the mixed benchmark

Program/Options	Instruction Count
`mc`, stack-based	274
`mc`, register-based	236
`mc`, register-based, optimized	213

Table 28: Assembly instruction counts for the mixed benchmark

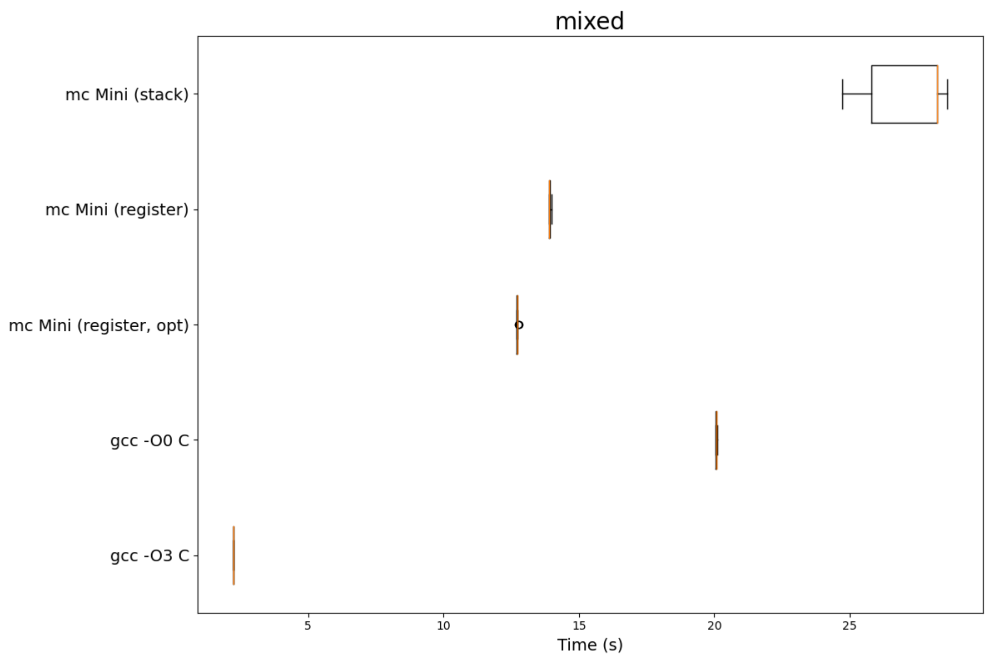


Figure 15: Runtimes for the mixed benchmark (lower is better)

3.16 OptimizationBenchmark

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	25.7536	0.0066
`mc`, register-based	18.1590	0.0506
`mc`, register-based, optimized	10.6336	0.0105
`gcc -O0`	24.2096	0.0085
`gcc -O3`	0.0189	0.0034

Table 29: Average runtimes for the OptimizationBenchmark benchmark

Program/Options	Instruction Count
`mc`, stack-based	1274
`mc`, register-based	931
`mc`, register-based, optimized	450

Table 30: Assembly instruction counts for the OptimizationBenchmark benchmark

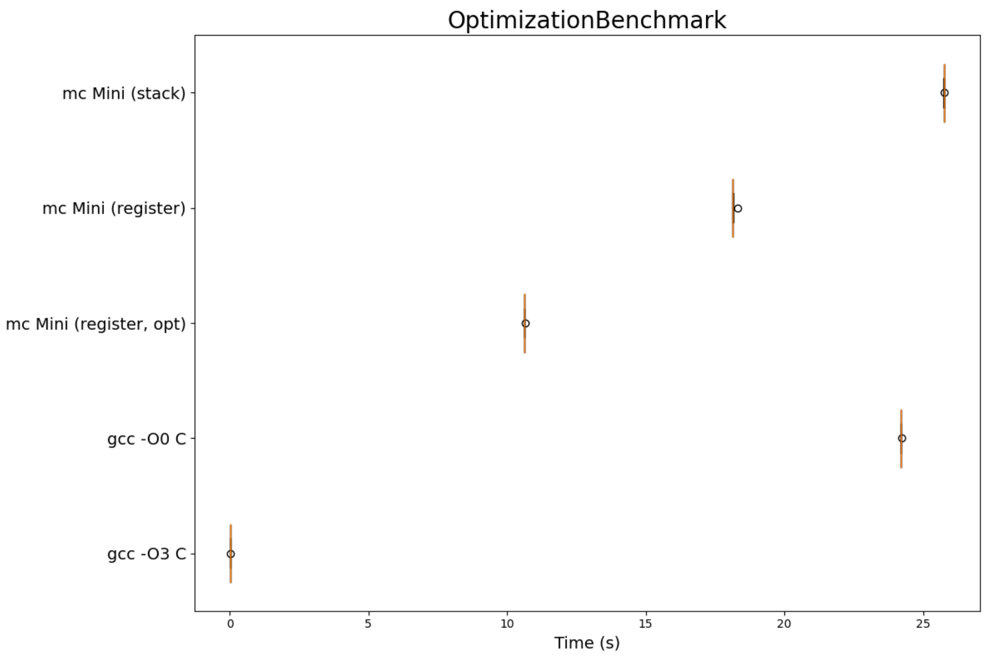


Figure 16: Runtimes for the OptimizationBenchmark benchmark (lower is better)

3.17 primes

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	16.5867	0.0104
`mc`, register-based	8.9620	0.0120
`mc`, register-based, optimized	6.9171	0.0099
`gcc -O0`	16.4468	0.0212
`gcc -O3`	4.4828	0.0121

Table 31: Average runtimes for the primes benchmark

Program/Options	Instruction Count
`mc`, stack-based	139
`mc`, register-based	131
`mc`, register-based, optimized	111

Table 32: Assembly instruction counts for the primes benchmark

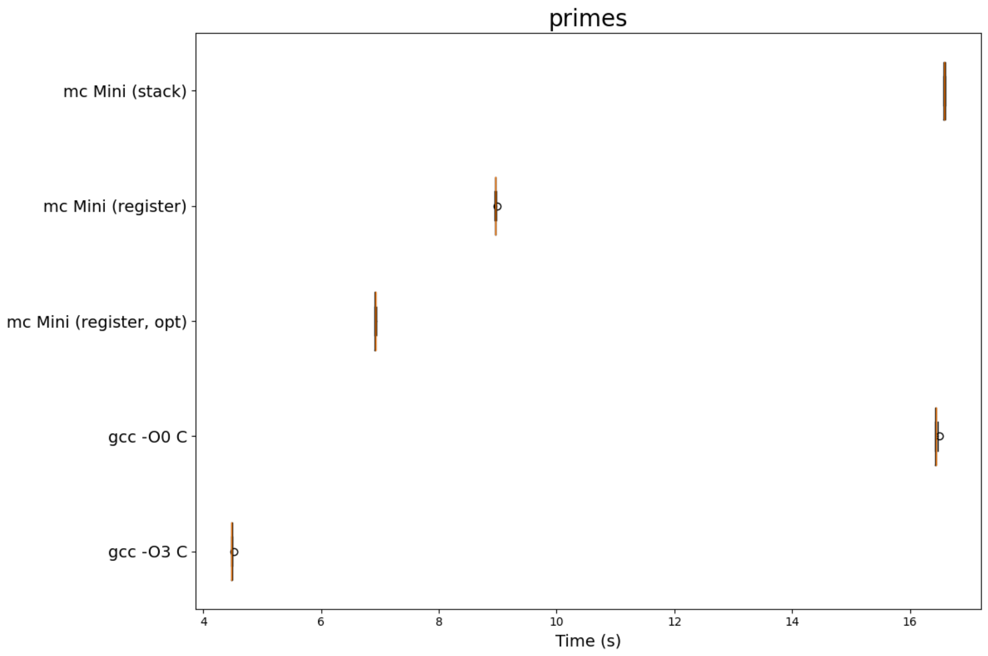


Figure 17: Runtimes for the primes benchmark (lower is better)

3.18 programBreaker

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0344	0.0122
`mc`, register-based	0.0328	0.0129
`mc`, register-based, optimized	0.0354	0.0121
`gcc -O0`	0.0350	0.0126
`gcc -O3`	0.0331	0.0130

Table 33: Average runtimes for the programBreaker benchmark

Program/Options	Instruction Count
`mc`, stack-based	136
`mc`, register-based	110
`mc`, register-based, optimized	106

Table 34: Assembly instruction counts for the programBreaker benchmark

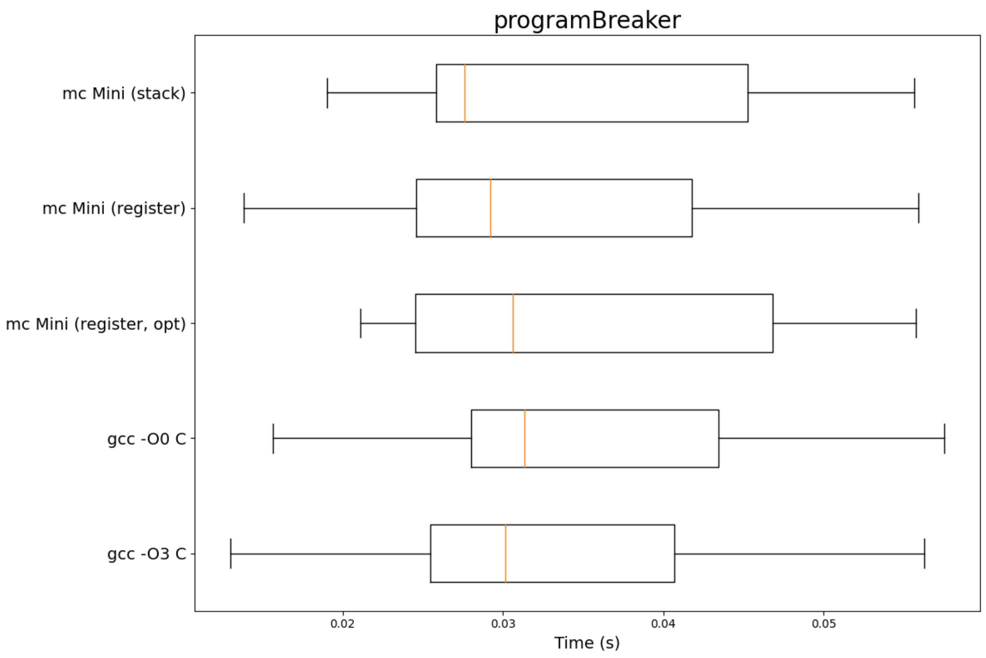


Figure 18: Runtimes for the programBreaker benchmark (lower is better)

3.19 stats

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	44.4924	0.0148
`mc`, register-based	14.9401	0.0097
`mc`, register-based, optimized	12.8263	0.0109
`gcc -O0`	42.3808	0.0143
`gcc -O3`	8.6023	0.0110

Table 35: Average runtimes for the stats benchmark

Program/Options	Instruction Count
`mc`, stack-based	309
`mc`, register-based	307
`mc`, register-based, optimized	273

Table 36: Assembly instruction counts for the stats benchmark

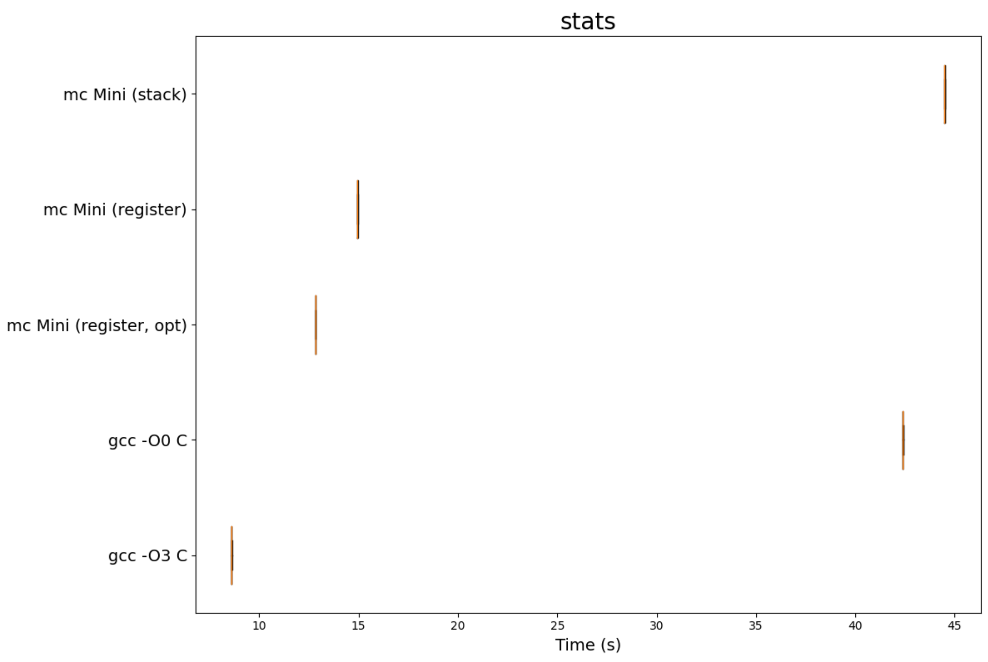


Figure 19: Runtimes for the stats benchmark (lower is better)

3.20 TicTac

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0101	0.0031
`mc`, register-based	0.0099	0.0042
`mc`, register-based, optimized	0.0101	0.0037
`gcc -O0`	0.0091	0.0043
`gcc -O3`	0.0092	0.0038

Table 37: Average runtimes for the TicTac benchmark

Program/Options	Instruction Count
`mc`, stack-based	554
`mc`, register-based	518
`mc`, register-based, optimized	426

Table 38: Assembly instruction counts for the TicTac benchmark

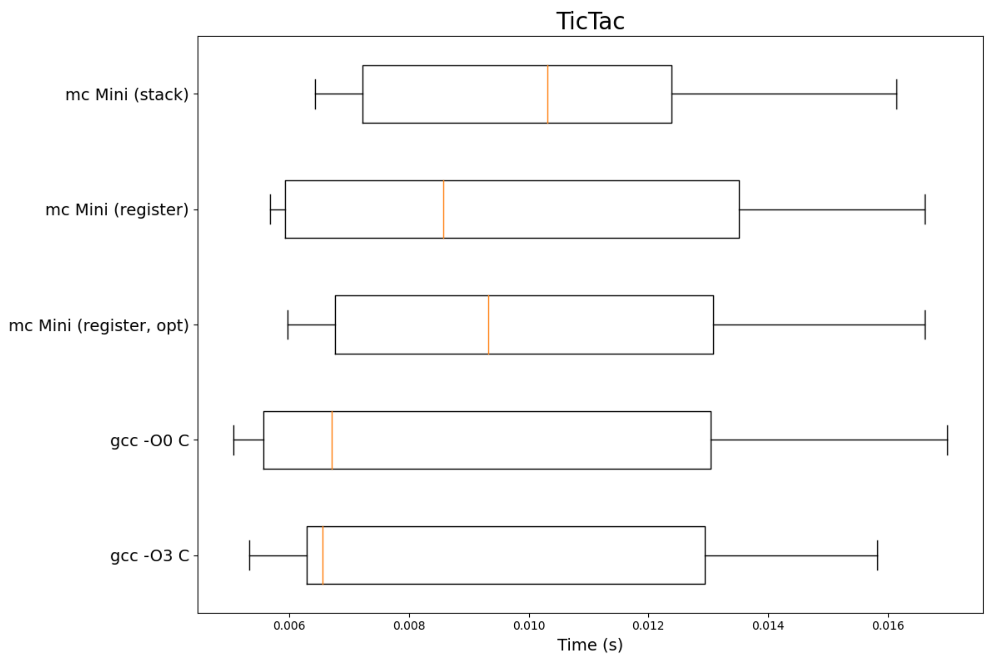


Figure 20: Runtimes for the TicTac benchmark (lower is better)

3.21 wasteOfCycles

Program/Options	Average Time (s)	Standard Deviation (s)
`mc`, stack-based	0.0118	0.0059
`mc`, register-based	0.0119	0.0057
`mc`, register-based, optimized	0.0116	0.0055
`gcc -O0`	0.0112	0.0068
`gcc -O3`	0.0124	0.0069

Table 39: Average runtimes for the wasteOfCycles benchmark

Program/Options	Instruction Count
`mc`, stack-based	76
`mc`, register-based	65
`mc`, register-based, optimized	59

Table 40: Assembly instruction counts for the wasteOfCycles benchmark

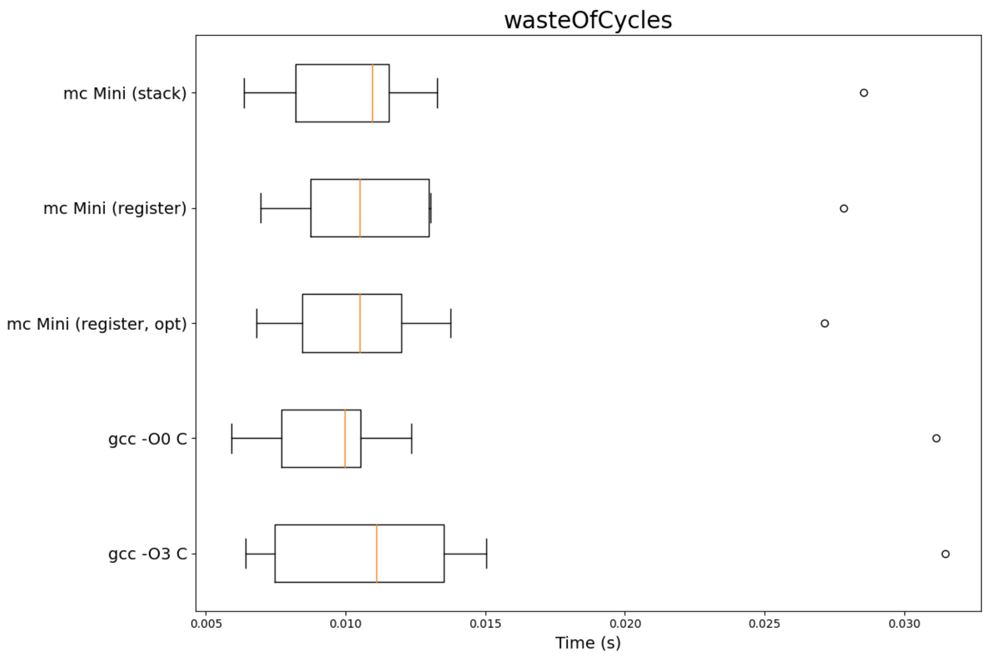


Figure 21: Runtimes for the wasteOfCycles benchmark (lower is better)