NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håvard Ramberg

Exploring compiler parallelism with Go

Master's thesis in Computer Science
Supervisor: Michael Engel
June 2022

Håvard Ramberg

Exploring compiler parallelism with Go

Master's thesis in Computer Science
Supervisor: Michael Engel
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The rise in popularity of general purpose parallel hardware, in the face of the power density wall, has transformed the way programmers create performant code. Much effort has been put into easily parallelisable, structured data access, SIMD operations, such as linear algebra. Less structured and erratic data formats have proven difficult to parallelise efficiently. I will be exploring the task of parallelising an elementary compiler using the Go programming language. I show that parallelising many of the compiler's internal passes decreases compilation time of a reasonably large source file by 36%. Transforming the AST into an SSA IR is beneficial for parallelism, because tree structures subverts parallelism by traditional means of traversing the tree. I show that smaller source files slow down compilation time when using multiple goroutines, as compared to larger files with many functions.

Lastly, I present a case that the Go programming language is a suitable replacement for the C programming language for an introductory compiler construction course at university level. The Go programming language offers built-in datatypes, like strings and hashmaps, which are essential components of a compiler. Additionally I show that a Go programmed compiler should have fewer lines of code than the C equivalent.

Acknowledgments

This work would not have been possible without the inspiration and aid generously offered by two lovely people. Firstly, I'd like to thank my supervisor, Professor Michael Engel. His feedback, constructive criticism and ideas have been invaluable for the completion of my thesis. Our many prolonged and faintly derailed supervision meetings have kept my mind at ease and spawned many interesting discussions within the field of computer science. Likewise, this work would seem implausible if not for former NTNU graduate Alf Martin Eggan. He has been a great asset for discussion and problem solving, and taught me elementary C programming language concepts when I first started my Master's study.

Contents

1 Introduction

Compilers are vastly interesting programs. Their general function is to translate some high level, human readable, programming language into architecture specific machine code [10]. Compilation time will depend on the system performing the compilation, the size and complexity of the source code and on the design and performance of the compiler software. Students probably experience the compilation process as a smooth and little time consuming process during their studies at university. However, the compilation process of large and complex software can last minutes, even hours [36, 37]. The gcc compiler toolchain has traditionally been the default C/C++ compiler for the Linux kernel [38, 39]. It is the single entry in the SPEC CPU suite that hasn't been removed or replaced since the suite's inception [34], suggesting that compile time is a valuable metric for computer performance.

Parallelism has been one of the key properties, and features of modern computer hardware, in increasing computer performance in the face of the memory wall and power density wall [34]. This has manifested itself as multi core CPUs and many core GPUs, enabling parallel execution. My contribution will be that of exploring parallelism in the *compilation process*. This is not to be confused by compilers that *transform* sequential code into parallel code [40]. Parallel compilation processes will throughout this thesis be the process of *compiling* source code into target code *using* one or more parallel worker threads to decrease the compilation time. The opposite is the traditional approach; a sequential, single threaded, compiler. I introduce the notion of *function level parallelism*, originating from an observation that functions are by definition independent code routines.

The Go programming language, a relatively fresh entry in the world of programming languages [25], is a high level programming language built for modern concurrent design. Since its release it has experienced growing popularity [24, 26, 27]. In this thesis, I'm examining if the Go programming language's ease of concurrency will enable function level parallelism to decrease compilation of an elementary programming language.

In addition to exploring compiler parallelism, this thesis aims to examine whether the Go programming language is a suitable replacement for the C programming language for an entry level compiler course at university level. The background and basis for this study is the spring 2021 edition of TDT4205 *Compiler Construction* course at the Norwegian University of Science and Technology (NTNU), where students were tasked with constructing an entry level compiler for an elementary programming language using the C programming language, Flex and Bison [16]. It was suggested by the former educator of the TDT4205 course that the Go programming language may prove more suitable for teaching compiler techniques because the Go programming language has more built-in features, such as hashmaps and automatic memory management (garbage collection), while still allowing the programmer to perform pointer programming and data type casting using interfaces. I show that a compiler written in Go could have fewer lines of code than a C, Flex and Bison equivalent.

Lastly I show that the Go programming language can utilise C-bindings to integrate a basic compiler frontend with the powerful LLVM toolchain to generate robust, and possibly optimised, binaries for potentially many target architectures. This feature isn’t pivotal in the research subject of the paper, but remains a proof-of-concept for future works.

The rest of this thesis is arranged followingly. The general concepts of a compiler is presented, followed by a brief definition of parallelism and how compilers can utilise function independency to enable parallel compilation. An elementary programming language is presented followed by a brief introduction to LLVM, LIR and hashtables. Chapter three presents experimental setup and general methodology. The results of the working compiler, benchmarks and verification tests are presented, highlighting which parts make the parallelisable compiler work in chapter four, Results. Chapter five, Discussion, analyses and discusses the findings and reasons for the given results. Chapter six presents similar work, before everything is wrapped up and concluded in chapter seven.

2 Theory

2.1 The compiler

A *compiler* is a computer program that translates a source program, written in a high level language, to architecture dependent machine code [10]. The process of translating the high level language to machine code is called *compilation*. A compiled program can be executed many times on different data. Examples of compiled source code languages are the C and C++ programming languages. It is also common for a compiler to report errors and diagnostics about the program being compiled [11].

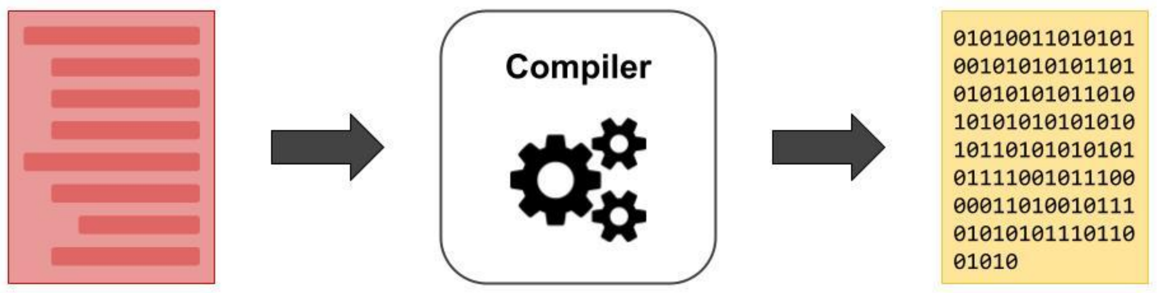


Figure 1: A compiler takes the source code on the left and translates it into the architecture dependent code on the right. The compiled program can be executed multiple times with different data.

A compiler is typically built into two or three *stages*. This composition is crucial for compatibility and compiler longevity. Fegaras illustrates this mathematically by suggesting that in order to support ***n*** source languages and ***m*** machine architectures one would have to create compilers for all permutations of source languages and machine architectures, which would total ***nm*** compilers [10]. With a two or three stage composition we can do with only ***n*** + ***m***. For the duration of this thesis I will be referencing a *three stage* compiler consisting of *frontend*, *intermediate* and *backend* stages. The compiler's performance is how quickly, in seconds, a compiler can compile a source program into target code.

2.1.1 Frontend

The first stage is called the *frontend*. The frontend translates the textual source language into a graph-like structure called a parse tree or *abstract syntax tree*, hereafter referred to as an AST. Translating the source language into AST requires scanning the source code for *tokens*, by grouping input characters [10]. This process is known as lexical analysis. A common way of scanning tokens is using finite state machines [10, 11, 14, 15]. A more recent suggestion is the use of a linear iterator and state functions [14]. Below is an example code. We define four tokens: *identifier (id)*, *integer (int)*, *assignment operator (=)* and *addition operator (+)*.

```
x = 1
y = x + 2
```

Code snippet 1: Two assignment statements. The latter assigns an arithmetic expression, the result of (x + 2) to the variable y.

The expected token stream of the above code snippet would be, from start to end of file:

id(x)
=
int(1)
id(y)
=
id(x)
+
int(2)

It is the job of the *parser* to request a token stream, as exemplified above, and translate the token stream into a data structure representing the initial source code [10]. This parsed data structure is called the parse tree or AST. Structured data collections (structs) with pointers is a popular way of expressing a graph/tree structure in computer science [46]. An example AST of the above token stream is shown below.

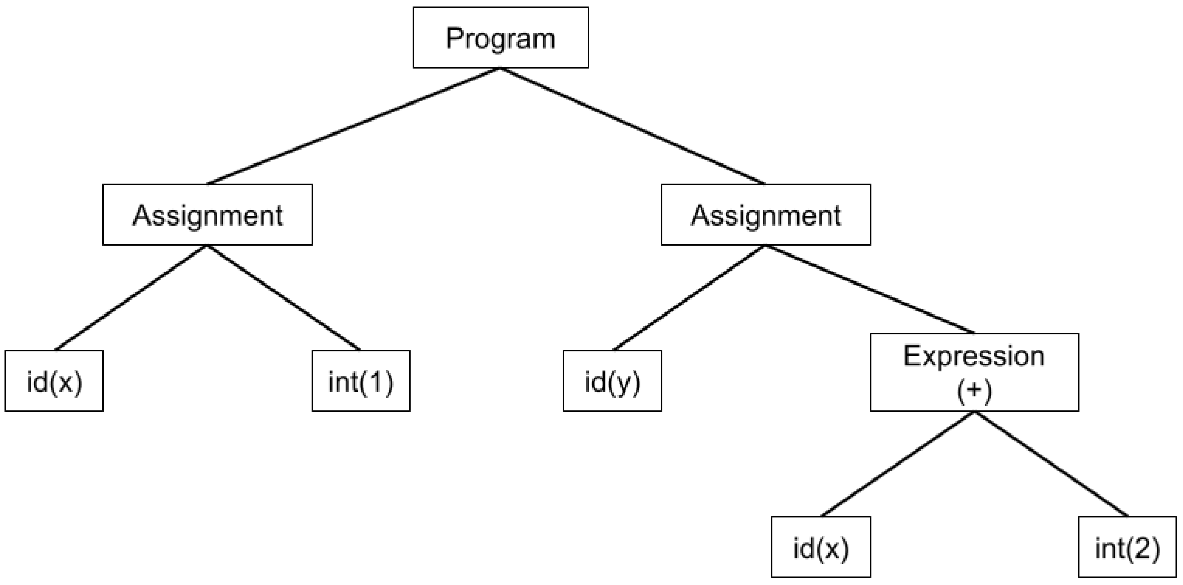


Figure 2: A parser has taken the token stream and constructed a graph, called the AST. The leaf nodes are called terminals, and map directly to the token stream nodes. Assignment, Program and Expression nodes are called non-terminals, and need not necessarily map directly to token stream elements.

Creating a valid syntax tree requires a pre-defined grammar. The parser uses the grammar and checks if the token stream matches one or more rules defined by the grammar. I won't delve deeper into grammars and parse tree theory for the remainder of this thesis, but the curious reader can have a look at [10], [11] and [32].

2.1.2 Intermediate

The second stage is the *intermediate stage,* which translates the syntax tree into an architecture independen *intermediate representation* (IR). As seen in [10], “This makes the task of retargeting the compiler to another computer architecture easier to handle”. Having a program representation that is both architecture and language independent enables great flexibility in language-architecture permutations, as stated in the previous chapter. Additionally it enables the compiler to perform language and architecture independent optimisations. Engel notes that optimisations are ways a compiler can “(...) maximise or minimise attributes of an executable program” [16]. Bennet suggests that the term optimisation in itself is somewhat misleading, because it is often undecidable when improvements lead to an attribute being optimised, and states that *optimisations* are in fact *improvements* [11]. This is supported by Engel, who suggests that some optimisations are NP-complete [16]. Nevertheless, the term optimisation will be used, as suggested by Bennet and Engel [11, 16]. There are numerous optimisations available. They include, but are not limited to, mathematical expression resolvement, loop-unrolling and live variable analysis.

Bennet suggests that three-address code (TAC) is a suitable IR for many optimisations, even stating that “It is recommended that compilers that are intended to perform much optimisation use TAC as their intermediate representation” [11]. TAC consists of two operands, a destination and an operator. A typical TAC textual representation is shown below.

TAC	Destination	Operand 1	Operand 2	Operator
`a = 1 + 6`	`a`	`1`	`6`	`+`
`b = a * 4`	`b`	`a`	`4`	`*`

Table 1: Three-address code usually constitutes a destination, two operands and an operator.

LLVM is a great example of a popular three-address code-like IR. Chapter 2.6 is dedicated to presenting the LLVM IR in more detail.

2.1.3 Backend

The third and final stage is the *backend* that optimises and translates the compiler IR into architecture specific code, or target code. Several types of target architecture specific code exist. Executable ELF-files is one example. While *assembler* is target specific, it is not executable. However, it is very near an executable target. For the remainder of this paper, assembler will be the target of the compiler, with one exception; the LLVM backend. By default the compiler will output aarch64 assembler, unless the compiler is told to use the LLVM backend and the system installed LLVM toolchain, at which point it will output aarch64 object code.

2.2 Parallelism

I want to start this section by first defining, and differentiating, the two commonly used computer science and computer engineering terminologies *concurrency* and *parallelism* for use in this Master's thesis. Goldman and Miller give the following definition of concurrency: "*Concurrency* means multiple computations are happening at the same time" [1]. There are multiple ways one can achieve concurrency in modern computer systems. Sharing CPU time among different processes is one. When multiple processes run for shorter periods of time, or time slices, on the CPU, they seem to be running simultaneously, but they share time on the CPU [2]. A more recent approach to concurrency is the introduction of multiple processor cores. This enables the CPU to genuinely run process threads concurrently on different processor cores.

Contemporary mobile, personal and server grade processors are often multicore processors. Multicore processors have introduced the possibility of *parallel* execution. Almasi and Gottlieb define parallel processor designs as a "(...) collection of processing that can communicate and cooperate to solve large problems fast" [4]. Rob Pike suggested that "Concurrency is about dealing with lots of things at once", while parallelism "(...) is about doing lots of things at once" [5, 6]. An example of concurrency could be the property of many modern computer systems being able to move the cursor while opening a large PDF file, or a server being able to process incoming data from a TCP connection while simultaneously listening to the same TCP connection for more data. Example of parallelism could be calculating the sum of two large vectors using multiple threads, each with their own portion of both vectors to add. The latter is an example of easy parallelism where little to no communication is required [4].

A very common way of achieving parallel execution is the use of *single instruction, multiple data* terminology (SIMD). SIMD means that a single centralised controller issues the same instruction unit to multiple execution units [3]. A very common usage scenario is the calculating dot product of two vectors. These operations are very basic, but provide excellent performance increase for structured data sets, such as arrays. They do not, however, offer means, by their very nature, although exceptions exist [3], to execute different instructions on different data simultaneously. *Multiple instructions, multiple data* (MIMD), uses multiple control units to issue different instructions simultaneously [3]. A MIMD-capable computer system has the ability to perform parallel execution on more general data that does not conform to some predefined, strict structure.

As such, *parallelism*, for the duration of this Master's thesis, is a MIMD-like operation executing on multiple processor cores, possibly doing *different* computations on different data, as will be explored in the next section; function level parallelism.

2.2.1 Parallelism in the Go programming language

The Go programming language introduces the concept of goroutines. Freeman states that "Goroutines are lightweight threads created and managed by the Go runtime (...) without needing to deal with the complications of operating system threads” [20]. Goroutines are lightweight threadlike structures because they are runtime structures, not operating system thread structures. The scheduling of goroutines implies that a goroutine is *not* equivalent to a traditional operating system level thread. Deshpande, Sponsler and Weiss suggest that a Go program may contain multiple operating system threads, yet the number of goroutines *should* outnumber the number of operating system threads created for the process [22]. Initially a Go program contains at least three goroutines: one for garbage collection, another for the runtime scheduler and, lastly, the user’s/programmer’s Go code. The Go runtime uses the *work-stealing scheduler* to schedule goroutines onto operating system threads, based on the number of processors available to the Go runtime [19, 23]. A work-stealing scheduler causes inactive processors to steal work from other processors, in the hopes that this causes even distribution of work tasks while minimising inter-processor communication [21]. If no processors are idle, no transfer of work between processors is issued. The work-stealing scheduler was specifically crafted for MIMD capable processor systems.

2.3 Function level parallelism

In mathematics functions map or define a rule or law between two values. Some describe functions as mapping an element in one set to exactly one element in another set [8] while others suggest that a free variable is mapped to a dependent variable [7]. An obvious observation is that functions take some inputs and outputs some results. The same is partially true for programming functions. A programming function, sometimes referred to as procedure, routine or sub-routine, is a set of self-contained instructions that perform a specific task [9]. A major difference between mathematical functions and programming functions is that mathematical functions always output the same result given the same inputs. In programming functions, this property is violated by the introduction of variables defined outside the scope of a function, referenced in the function and not given to the function as an input.

The above is true for executing functions at run-time. At compile-time however, functions are independent structures, because they represent a self-contained set of instructions. Given a simple C code example; we define two functions *add* and *sub*, both which takes two integers and adds and subtracts the two integers parameters respectively.

```c
int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}
```

Code snippet 2: Two C-styled functions that are completely independent. Neither add references sub, or vice versa.

The above functions are completely independent. Adding a third function, *foo*, which calls both above functions, will make function *foo* dependent on both functions *add* and *sub*.

```
int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int foo(int a, int b) {
    int c = add(a, b);
    return sub(c, a);
}
```

Code snippet 3: Function foo calls both add and sub, making foo dependent on the existence and function headers of functions add and sub.

The above example proves a property of programming languages that violates the previous claim that functions are independent at compile-time. It is necessary to *adjust* the initial statement *somewhat*. Functions defined in the same scope, be it local or global, are independent at compile-time, if we can resolve all dependencies declared in the same scope or superscope(s) at compile-time.

It is not relevant for function *foo* to know *what* functions *add* and *sub* do, it only needs to know that they exist and their respective parameter(s) and return type(s). It is my proposition that functions *can be compiled independently*, because in order to compile functions independently we have to know that a function, or multiple functions, exist in the first place. Following this observation we can easily conclude that function references to other functions in function bodies can be looked up at compile-time, given that we first know all function definitions in the current scope. If we can generate functions independently at compile time we can also generate functions in *parallel*.

Things get more complicated when we introduce external libraries and function definitions and multiple files, but, as will be explained in the next chapter, this is not a problem this thesis has to solve.

2.4 The very simple language

Very simple language (VSL), defined in Jeremy Peter Bennet's book Introduction to compiling techniques : a first course using ANSI C, Lex and YACC, is an elementary programming language [11]. Bennet states that VSL was designed to be "(...) a simple block-structured procedural language, with assignment statements, while loops, if-then-else branches and simple expressions" [11]. The full VSL specification is not provided in this thesis, but can be found in Bennet's book.

2.4.1 Alterations to VSL

In Bennett's original VSL definition he writes "(...) the null_statement does nothing" [11]. Some confusion must have been introduced when Bennet's *null_statement* evaluates to a CONTINUE, which at some point someone must have believed to be the commonplace continue-statement within loop-bodies of modern programming languages. This misunderstanding certainly manifested itself in the NTNU TDT4205 course's Practical Exercise 6, where the very last task writes "Implement conditionals, while loops and continue" [16]. Because implementing the modern continue-loop statement requires only the addition of a stack to keep track of loop scopes, it should be considered trivial to implement this feature in the VSL language. Bennett also notes that expressions are "(...) evaluated with their conventional arithmetic meaning using unsigned arithmetic" [11]. For reasons unknown, integers were signed during the NTNU TDT4205 spring 2021 course [16].

2.5 Typed VSL

VSL is an elementary programming language, fit for third and fourth year computer science students. It is my desire to extend the VSL language. The major change introduced in this thesis is the introduction of floating point data variables and expressions. VSL initially had a single data type, unsigned integer, and didn't need the concept of data types. I name this new extended language the *typed VSL* language. In typed VSL arithmetic expressions are evaluated as either 32-bit single or 64-bit double precision floating point (IEEE 754) or *signed* integer expressions. One new data type is added to accommodate floating point numbers: the *float* type. The existing integer type is named *int*. Types are statically declared by appending declarations with either data type. Functions declare their return type by appending a data type after the parenthesised parameter list. See Table 1 below for examples.

	VSL	**Typed VSL**

Single declaration	`var x`
	`var y`
		`var x int`
		`var y float`

Multiple declarations	`var a, b, x, y`	`var a, b int`
		`var x, y float`

Function declaration	`def sqr(a)`	`def sqr(a int) int`
Function declaration with multiple parameters	`def foo(a, b, x, y)`	`def foo(a, b int, x, y float) float`

Table 2: Comparison of VSL and typed VSL for variable and function declarations. The typed VSL borrows styling and idioms from the Go programming language.

2.6 LLVM

LLVM is a popular intermediate representation used by some compilers. It is a statically typed TAC like IR that supports a wide range of datatypes, such as signed and unsigned integers, floating point numbers, pointers, structures, arrays and variable lists [41]. When defining data, the programmer or compiler has to supply the data type. LLVM is a static single assignment (SSA) IR. SSA replaces every variable with a new variable that is defined only once [29]. LLVM calls these distinctly defined variables *virtual registers*. When LLVM generates the target architecture code, the virtual registers are allocated a hardware register by some register allocation algorithm. A simple C-style main function with the printing of the traditional “Hello world!\n” message is shown below in textual LLVM syntax.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello world!\0A\00", align 1

define i32 @main(i32 noundef %0, ptr noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca ptr, align 8
  store i32 0, ptr %3, align 4
  store i32 %0, ptr %4, align 4
  store ptr %1, ptr %5, align 8
  %6 = call i32 (ptr, ...) @printf(ptr noundef @.str)
  ret i32 0
}
```

Code snippet 4: An example of LLVM textual representation. The main function declares three variables on the stack, using the alloca statements. Virtual register %3-%5 hold the pointers to these stack variables, which later be loaded from memory, or stored to.

A virtual register is denoted by *%<number>*. They are local to function scopes. Globally declared constants, variables and functions are prepended by *@*. The resemblance to assembler code is striking.

2.7 LIR

Lightweight intermediate representation (LIR) is an elementary SSA based IR that aims to simplify compiler IR, making it usable for an entry level compiler course. LIR is greatly inspired by LLVM. The major differences are outlined in the table below.

	LIR	**LLVM**
Data types	Signed integer, floating point	Signed/unsigned integer, single, double and quad precision floating point, structures, booleans, strings, variable length arguments, pointers, arrays
Bitsize of data types	Target architecture dependent	Specified in IR generation
Supported targets	aarch64	aarch64, RISC-V 32/64, x86 and many more
Optimisations	None	Dead code elimination, mem-2-reg, loop-unrolling and many, many more
Written in	Go	C++, assembler

Table 3: Comparison of LIR and LLVM.

LIR consists of, similarly to LLVM, a module that has some globals and some functions, where each function has some parameters and basic blocks. Basic blocks hold instructions. An instruction generally falls into one of three main categories: Data instructions, Memory instructions and Branch instructions. Data instructions and the Memory instruction Load generates a result that is put in a *virtual register*. Branches and store instructions do not generate results.

LIR offers two types of data types: Int and Float. They are target architecture dependent, meaning that bit size, and subsequently floating point precision, is decided upon target architecture code generation. LIR automatically casts Int to Float when computing expressions where one operand is a Float data type and the other operand being the Int data type. The Int type supports all arithmetic operators (+, -, * and /), bitwise masking (&, ^ and |), bitwise shifting (<< and >>), arithmetic negation (-) and bitwise negation (~), whereas Float only supports the four basic arithmetic operators (+, -, * and /). During target architecture code generation the Int and Float data types are assigned the maximum allowed bit size, given target architecture. For 32-bit architectures Int becomes a 32-bit signed integer, while Float becomes 32-bit IEEE-754 single precision floating point. When targeting a 64-bit architecture Int becomes a 64-bit signed integer, whereas Float becomes IEEE-754 64-bit double precision floating point.

2.8 Hash tables

A hash table is a data structure that arranges *key-value pairs* in a dictionary style data structure that supports the methods *insert*, *search* and *delete* [12, pp. 253–285]. A hash table contains an underlying array and a method for generating an array index from the key by computing. It is popular to compute a hash of the key using a *hash function*, denoted *h(k)*, where *k* is the key. A hash function takes an arbitrary byte stream and generates a fixed length output based on the input, typically an integer for indexing the underlying hash table array [13]. This enables lookups into the array with data types other than integers, such as strings.

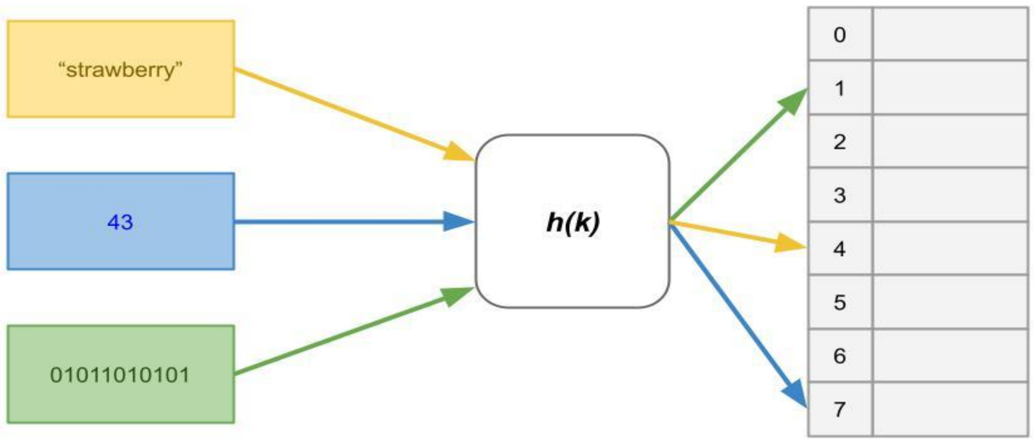


Figure 3: A hashtable that accepts strings, integers and arbitrary bytes as input keys and indexes the underlying table using the hash function *h(k)*.

A problem encountered with hash tables are *collisions*. Collisions occur when two keys, k_1 and k_2, where $k_1 \neq k_2$, but $h(k_1) = h(k_1)$ [12, pp. 253–285]. This causes two different inputs to be mapped to the same index in the hash table.

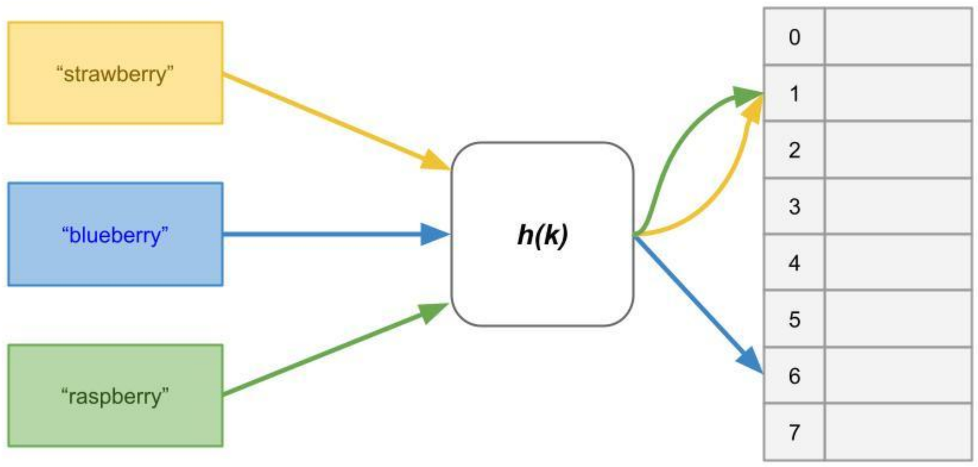


Figure 4: "strawberry" and "raspberry" are hashed to the same index by hash function *h(k)* in this example. This is problematic because the table entry at index 1 can only store one key-value pair.

There are multiple ways of resolving collisions. One way is to chain the stored key-value pairs in linked lists [12, pp. 253–285]. Linking incurs a cost of traversing the list until the entry with the correct key is found.

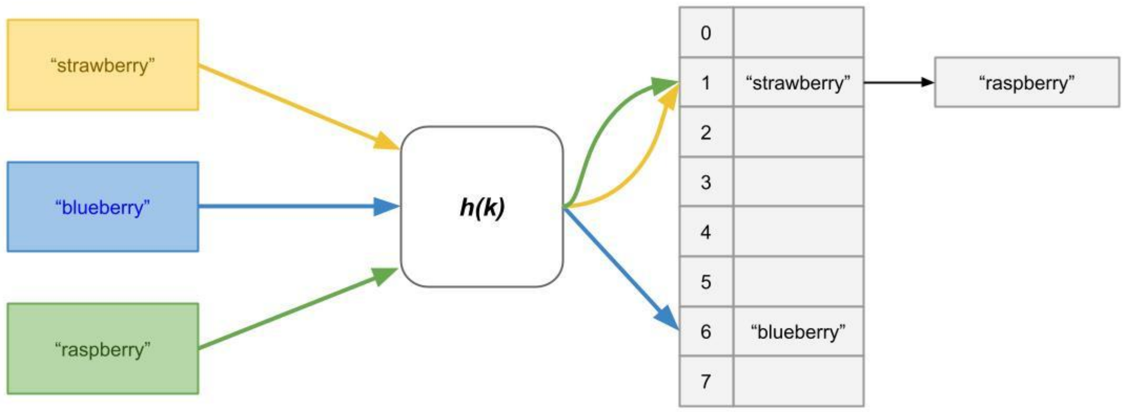


Figure 5: By linking the entry for "raspberry" with the entry for "strawberry" we have resolved the collision caused by the hash function *h(k)*.

Another way is the bucket system, where the hashed index selects a sub array that can hold multiple key-value pairs. These buckets can be chained once more if they become full, as is done in the Go programming language hashmap implementation [13].

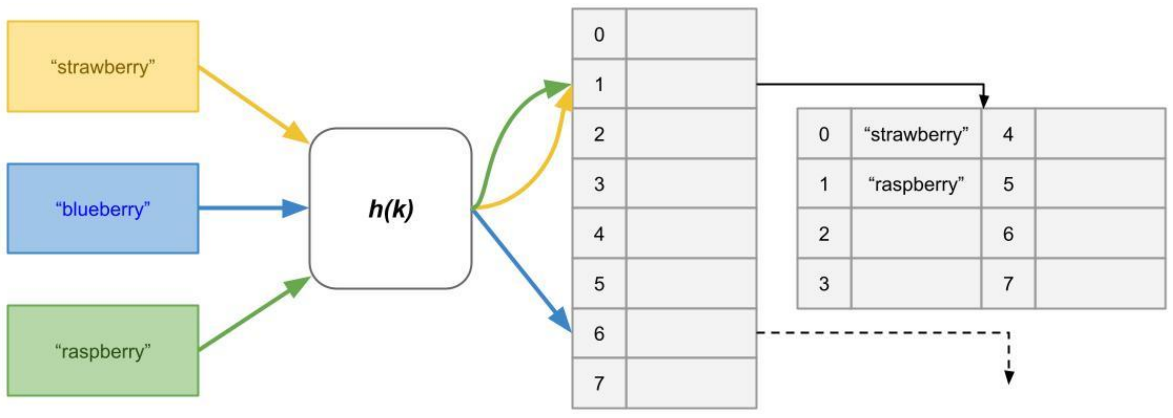


Figure 6: The hashed index points to a new underlying array called a bucket. This bucket holds the data, but must be searched linearly for the correct key-value pair.

Hash tables are useful as compiler symbol tables because they enable fast lookups using string identifiers [12, pp. 253–285]. The almost constant time cost of generating the hash index, instead of iterating an array or linked list containing *all* entries, results in speedy lookups.

2.9 The typed VSL compiler

All of the previously mentioned topics coalesce into the VSL compiler, named *vslc*. The compiler is programmed in Go, using as many built-in features from the Go programming language, such as dynamically sized arrays (slices), hashmaps, strings, goroutines and channels. The parser is generated using the *goyacc* parser generator [48]. By default vslc outputs aarch64 assembler, using the LIR IR. The *-ll* flag can be passed, instructing the compiler to use the system installed LLVM toolchain to generate aarch64 object code. All stages past the parser are fully parallelisable, with the exception of the LLVM toolchain, by passing the *-t* flag, followed by the desired number of goroutines, to the vslc executable.

3 Methodology

Performance benchmarks were primarily run using Go's built-in testing and benchmarking framework [56]. An alternative would be to use either the bash time or Unix time programs. However, the latter two provide very low sub-second fidelity [58]. Go's testing framework on the other hand was built for high fidelity timing, providing nanosecond reporting by default. Results are reported in nanosecond per loop iteration by default. Number of loop iterations is decided by the go test framework. The number of iterations is adjusted automatically by the framework, but benchmarks execute for a minimum of one second by default [57].

3.1 Hash comparison procedure

Symbol tables are a vital part of a compiler for looking up variable identifiers and functions. Assessing hash table performance gives insight into which symbol table solution offers the highest performance. Previous renditions of the TDT4205 course used a CRC32 based hash table [16]. This thesis pits multiple CRC32 implementations, a DJB2 implementation and the Go programming language's hashmap head-to-head. A hash table supports three operations; *insertion*, *lookup* and *deletion*. Lookup is the most frequent operation, thus lookup performance is emphasised the most. Benchmarks were written in the Go programming language using Go's built-in *testing* package. See Appendix A for hashmaps benchmarking source code. Five hashtable candidates were benchmarked; three CRC32 based, one DJB2 and finally Go's hashmap. Each hashtable candidate is given a callsign.

Callsign	**Algorithm**	**Description**
crc32p	CRC32	Direct port from the TDT4205 spring 2021 C version CRC32 hash table. Collision entries are *prepended* at the beginning of a linked list.
crc32re	CRC32	Re-implementation of the CRC32 port in Go using interfaces and linked lists. Collision entries are *appended* at the end of the linked list.
crc32opt	CRC32	Optimised version of CRC32 re-implementation. Hardcoded support for symbol table structs and uses 8 pair bucket tables with linked chaining similar to Go hashmap.
djb2opt	DJB2	Similar to CRC32 optimised, but uses DJB2 hashing algorithm.
goh	AES based (?)	The Go programming language built-in hashmap.

Table 4: Hashtable candidates listed. The candidates are later referred to by their callsigns (first column).

3.2 Parallel compiler benchmark procedure

The parallel benchmark suite is constituted by the five benchmarks listed below. Neither benchmark measures the time it takes to read the file contents or write to the destination file. All write operations from the perspective of worker goroutines are called, but not executed by the mockup writer. The worker goroutines' output memory contents get deallocated, not written to file.

Benchmark name	**Description**
Aarch64	Benchmarks all compiler stages from source string to writing assembler to the mockup output writer.
ASTOptimisation	Benchmarks optimising the parse tree into a more minimalistic AST. Mainly recursive list flattening, lonely node deletion and arithmetic constant expression resolvement.
LIRGeneration	Benchmarks transforming the minimalistic AST into LIR SSA.
RegisterAllocation	Benchmarks the register allocation algorithm for the LIR SSA for a given target architecture (aarch64 in this thesis).
AssemblerGeneration	Benchmarks transforming the LIR SSA into assembler and write to mockup writer.

Table 5: Listing of all parallel compiler benchmarks in the parallel compiler benchmark suite.

Four source files were selected from the */resources/vsl_typed* directory of the source code of Appendix B. These four source files differ in size, complexity and number of defined functions.

1. *aamanyfuncs.vsl*, is a large ~1,000 line source file with 12 defined functions
2. *harder.vsl* is a relatively small (~30 lines of code) file with two functions
3. *if2.vsl* is a very small (~10 lines of code), single function, source file
4. *multi_hello.vsl* is a medium/small source file of ~40 lines of code with three defined functions

Although there are 28 source files available in the */resources/vsl_typed* directory of Appendix B, only four were selected. This decision was made on the basis of avoiding cluttering of figures, general readability and avoiding overwhelming the reader, and writer, with numerous data sets and statistics.

3.3 Experimental setup

Hashmap comparison tests and benchmarks were executed on a Lenovo T430 laptop, model ThinkPad 2347GU8 running an Intel Core i5-3320M processor with hyperthreading [31] and 8GB of DDR3 main memory, running Manjaro Linux 21.2.1 (Qonos). The Go version was go1.15.2 linux/amd64. The power lead was inserted and all other desktop applications were fully terminated during benchmarks.

A second hardware platform was added to provide more CPU cores for parallel compiler benchmarks. This second platform was an HPE ProLiant DL360 G7 server with two four core Intel Xeon E5630 processors with hyperthreading [30], 16GB of DDR3 memory running the VMWare ESXi 6.5 bare metal hypervisor. A virtual machine with all 16 available vCPU cores, 2GB of main memory and running Centos 8 Stream, with kernel Linux 4.18.0-383.el8.x86_64, was allocated for the parallel compiler benchmarks. All other virtual machines on the hypervisor were shut down. The below table summarises the two benchmarking platforms.

Callsign	**Hardware**	**CPU**	**RAM**	**OS**
laptop	Lenovo T430	Intel Core i5-3320M	8GB	Manjaro Linux
vm	ESXi 6.5 virtual machine on HPE DL360 G7	16 vCPUs, (2x Intel Xeon E5630)	2GB	Centos 8 Stream

Table 6: This table lists the hardware specifications for the laptop callsign system, used for benchmarking hash algorithms, and the virtual specifications for the vm callsign system, used for benchmarking the parallel compiler.

The hashmap benchmark was executed on the *laptop* callsign configuration only, whereas the compiler parallel benchmarks were executed on *vm* callsign configuration only.

4 Results

In this chapter I present the data driven findings of my thesis. The results of a simple hashmap implementation test is presented first, followed by the benchmarks of the parallel compiler. Lastly I present a simple line count comparison of the 2021 edition C, Flex and Bison compiler from the TDT4205 course and the parallelisable compiler introduced in this thesis.

4.1 Hash comparison

See Appendix B for hash comparison source code. There were 10 test entries that were inserted into the hash table. The results of the hashmap benchmark are shown below, and found in raw format in Appendix D.

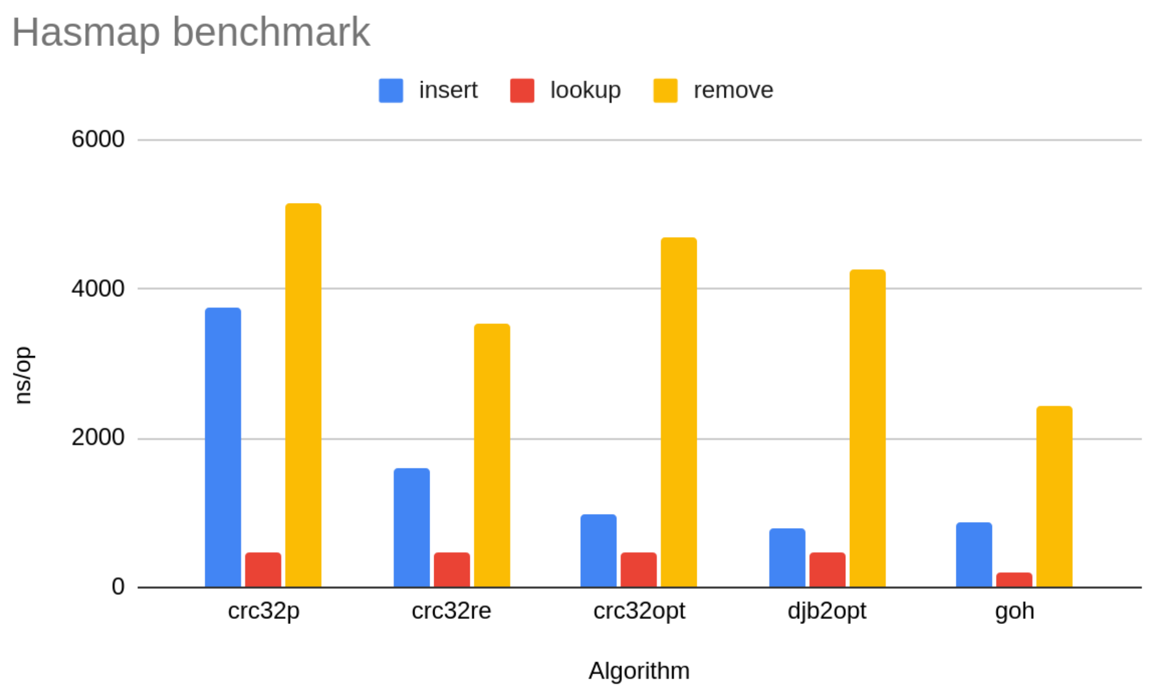


Figure 7: The lookup operation (red) is the most performance critical property of a hashmap used as a compiler symbol table. Insert operation (blue) and remove (yellow) operations are not performance critical. Lower ns/op is better.

The Go language built-in hash map (*goh*) outperforms both CRC32 hash table implementations. It sees the greatest performance advantage in lookup operations, which is common in a symbol table setting. It performs lookups 130.0% faster than *djb2opt*, *crc32opt* and *crc32re* respectively, and 133% faster than *crc32p*. The Go hashmap experiences less speedup in the *remove* operation compared to one of the CRC32 implementations, *crc32re*, but it is still considerably faster. When we look at the *lookup* benchmark, by far the most performance critical property of a symbol table, we see that the built-in Go hash map is more than twice as performant as three other candidates. It appears that even the optimised *crc32opt*, which works in similar fashion to *goh* using bucket tables, and using pre-defined data types, does not considerably improve lookup times than the other two CRC32 candidates.

4.2 Parallelisable compiler

There were a total of 28 VSL source files and five benchmarks in the parallel compiler benchmark suite. If all benchmarks were to be run for compiler parallel goroutine count of 1 through 16, it would total 2240 benchmarks. Clearly it would be inadvisable to cram all 2240 results into one or more diagrams. The interested reader may find all benchmark results data in the */doc/bench.csv* file of the source code provided by Appendix B and raw data in Appendix C.

Appendix B additionally provides the */doc/bench_old_all_missing_some.ods* OpenDocument Spreadsheet file that has many more datasets for most of the 28 source files. The selected results are chosen to highlight both the great and grim results, as well as establishing a trend among the majority of the results. Benchmark source code is found in Appendix B */src/vslc_test.go*. The result metric, *ns/op*, indicates average time per benchmark iteration. Lower ns/op is better.

We begin by examining the *total compilation time*, from source string to writing the result assembler to a mock writer, as demonstrated by the Aarch64 benchmarks.

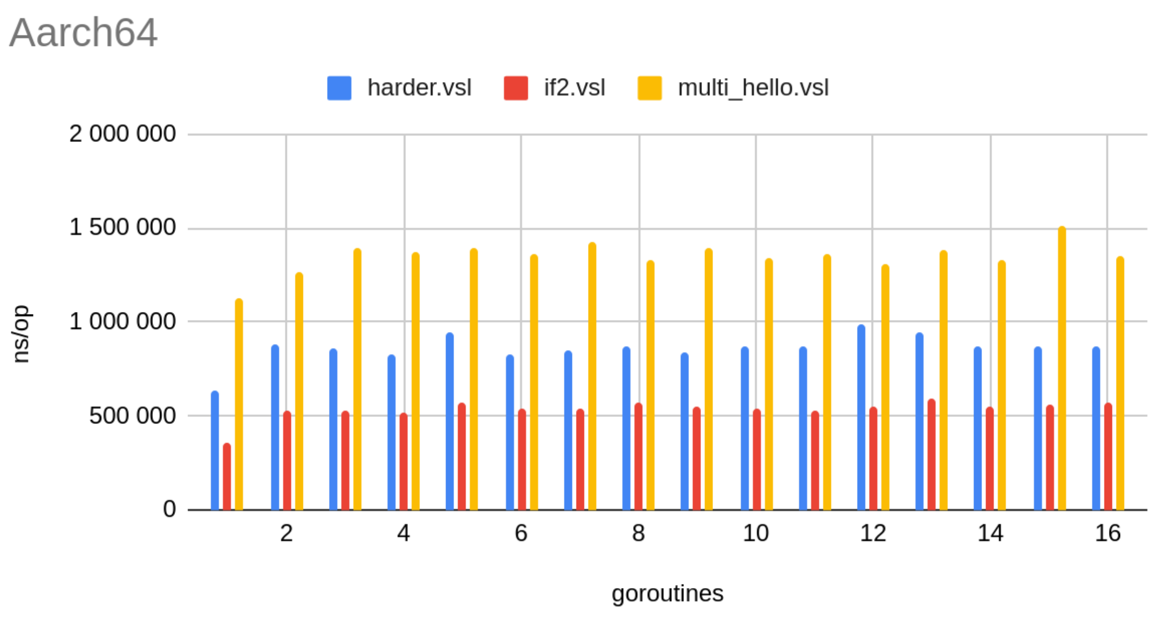


Figure 8: All three source code files, harder.vsl, if2.vsl and multi_hello.vsl, experience an increase in compilation time when the switch to multiple goroutines was made. Sequential run has goroutine count equal to one. Lower ns/op is better.

The overall compilation time of the three smaller files *harder.vsl*, *if2.vsl* and *multi_hello.vsl* suggest a grim trend for the parallelisable compiler. The relative *slowdowns* compared to the sequential compiler, where goroutines equals one, using two goroutines are 28.0%, 31.8% and 11.1% for the *harder.vsl*, *if2.vsl* and *multi_hello.vsl* files respectively.

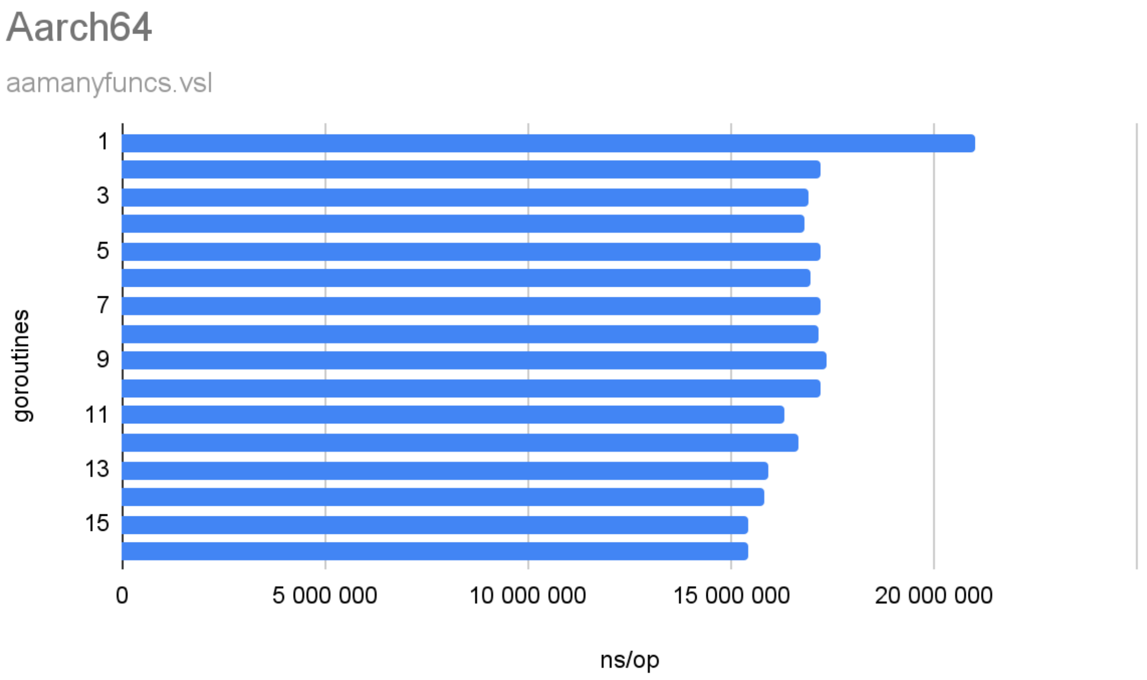


Figure 9: The largest source file, *aamanyfuncs.vsl*, with multiple large function definitions experiences a decrease in compilation time when introducing multiple goroutines. Lower ns/op is better.

aamanyfuncs.vsl suggests that larger source files with many long function bodies benefit slightly from function level parallelism. Using two worker goroutines decreases compilation time by 22.0%, compared to the sequential compiler. The greatest speedup is achieved by 15 and 16 goroutines, resulting in a 36.0% speedup. This is surprising, because I'd expect the greatest speedup to be at 12 goroutines, because the *aamanyfuncs.vsl* file defines 12 functions.

The four subsequent benchmarks attempt to detail the cause for the aforementioned overall compilation times. The first parallelisable process of the compiler happens at the frontend-intermediate boundary, with the AST optimisation pass.

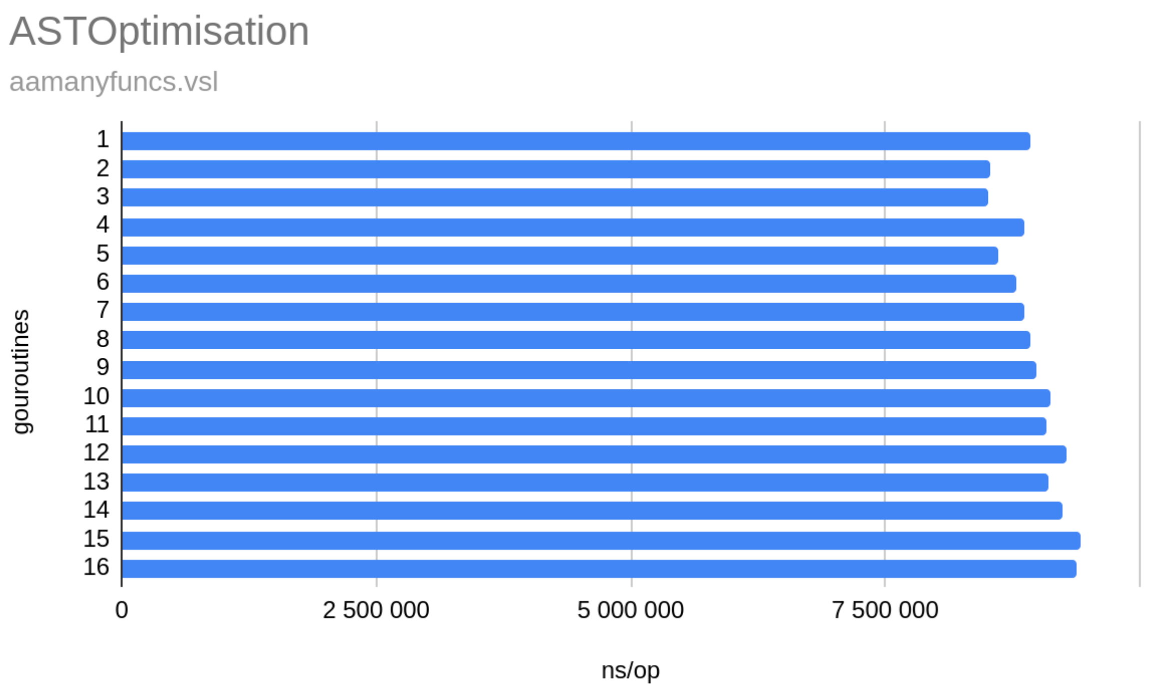


Figure 10: The parallel AST optimisation pass doesn't noticeably increase performance, but demonstrates the opposite effect of slowing the process down. Lower ns/op is better.

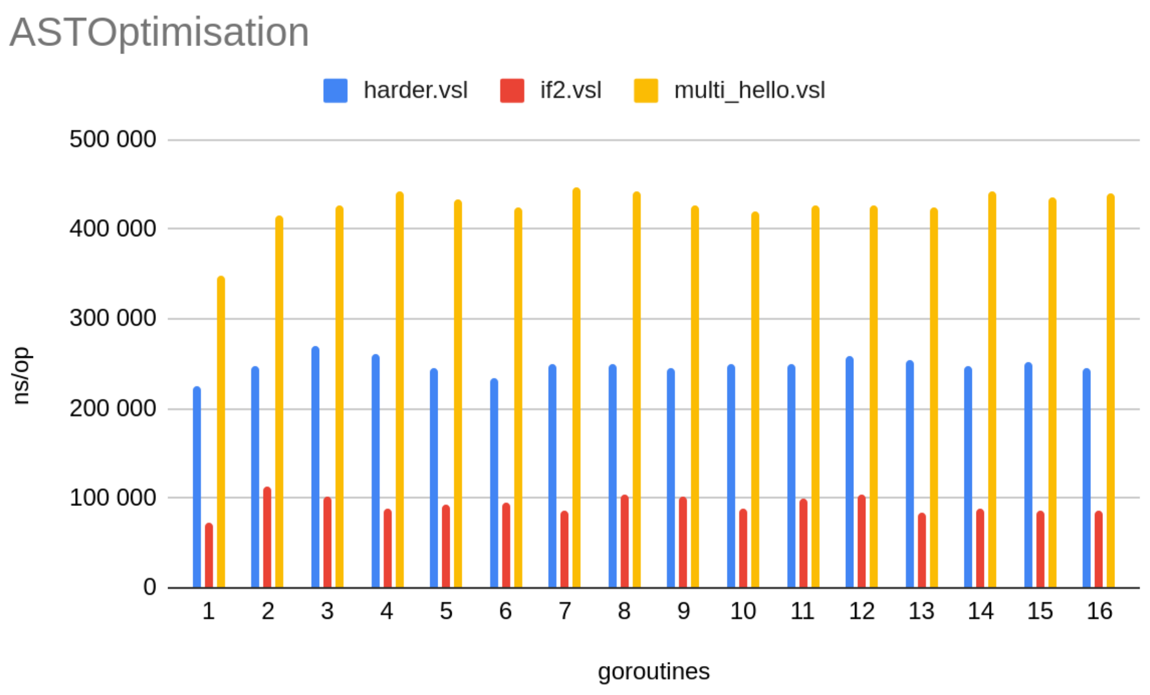


Figure 11: The figure illustrates how smaller file size source files slow down when performing AST optimisations. Lower ns/op is better.

Strangely, *aamanyfuncs.vsl* doesn't experience a similar execution time decrease during the AST optimisation pass. Performance increases slightly for two, three and five worker goroutines, but nowhere near a reduction nearing half or two thirds as would be expected by easily parallelisable tasks. The three remaining source files retain a nearly constant compilation time, irrespective of goroutine count. However, all parallel runs have higher execution time than their respective sequential runs. This compiler pass takes the longest time to complete of all the four individual passes.

Translating the AST into LIR SSA representation is a task that is not necessarily easily parallelisable. The AST sub-trees remain independent, but stalling is expected because of synchronising barriers for global definitions before the goroutines proceed to transform the function bodies. Additionally, there are mutex locks for operations that change the state of the global module, such as adding string literals or integer or floating point constants.

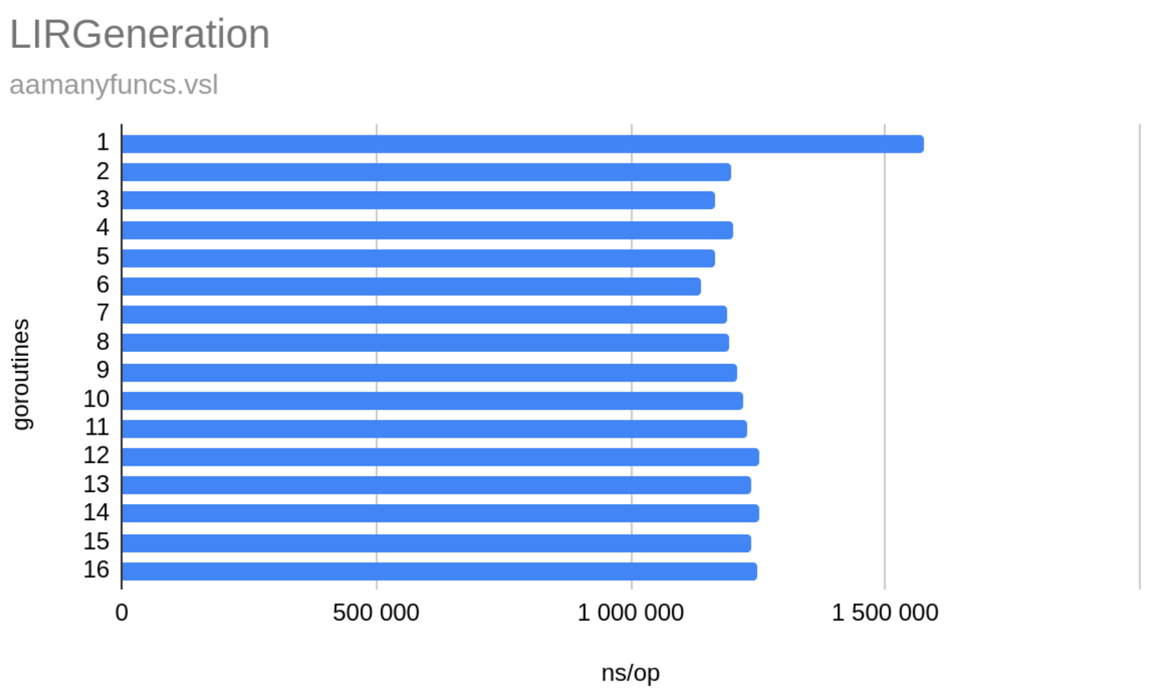


Figure 12: The figure shows that LIR generation is somewhat parallelisable, but there are diminishing returns for applying many goroutines. Lower ns/op is better.

The larger *aamanyfuncs.vsl* source file displays speedup similar to the overall Aarch64 benchmark. There is a noticeable 31.6% speedup when going from one worker goroutine to two, but the speedup from there on after is negligible.

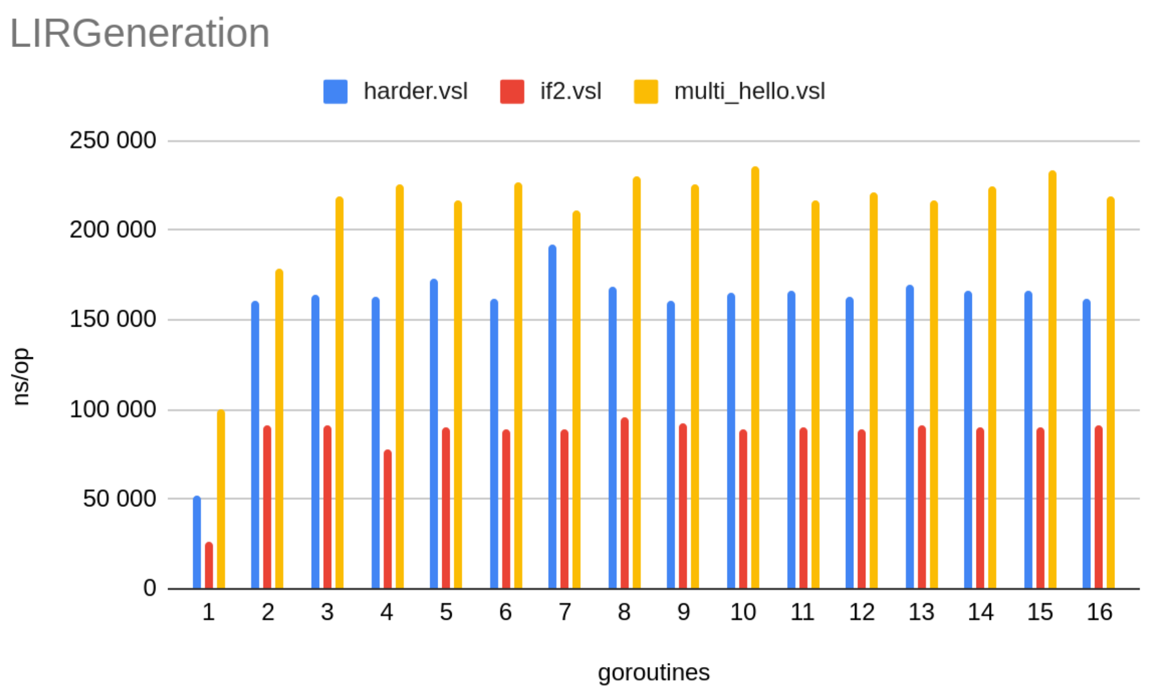


Figure 13: The figure illustrates the slowdown of many goroutines and small file sizes. Lower ns/op is better.

The grim trend of slowing down continues for the smaller source files. The slow down is 67.8%, 70.7% and 43.6% for the *harder.vsl*, *if2.vsl* and *multi_hello.vsl* respectively executing with two worker goroutines compared to one.

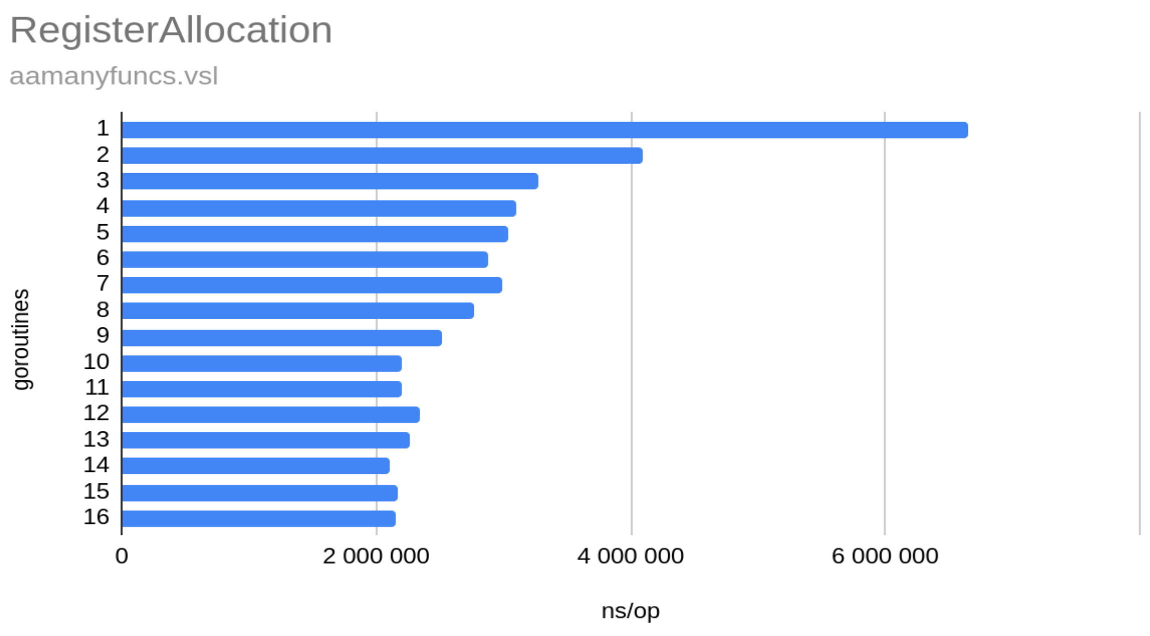


Figure 14: Register allocation for the *aamanycunfs.vsl* source file resembles the expected result of performing highly parallelisable tasks with more goroutines. Lower ns/op is better.

Register allocation experiences dramatic performance increase when parallelised for the aamanyfuncs.vsl source file. This is the result I would expect from a highly parallelisable task; each added goroutine should reduce the allocation time by a fraction. Although we do not see perfect *1/n* scaling, the compiler boasts a 62.7% performance improvement for two versus one goroutine and a considerable 215.3% performance increase with 14 goroutines.

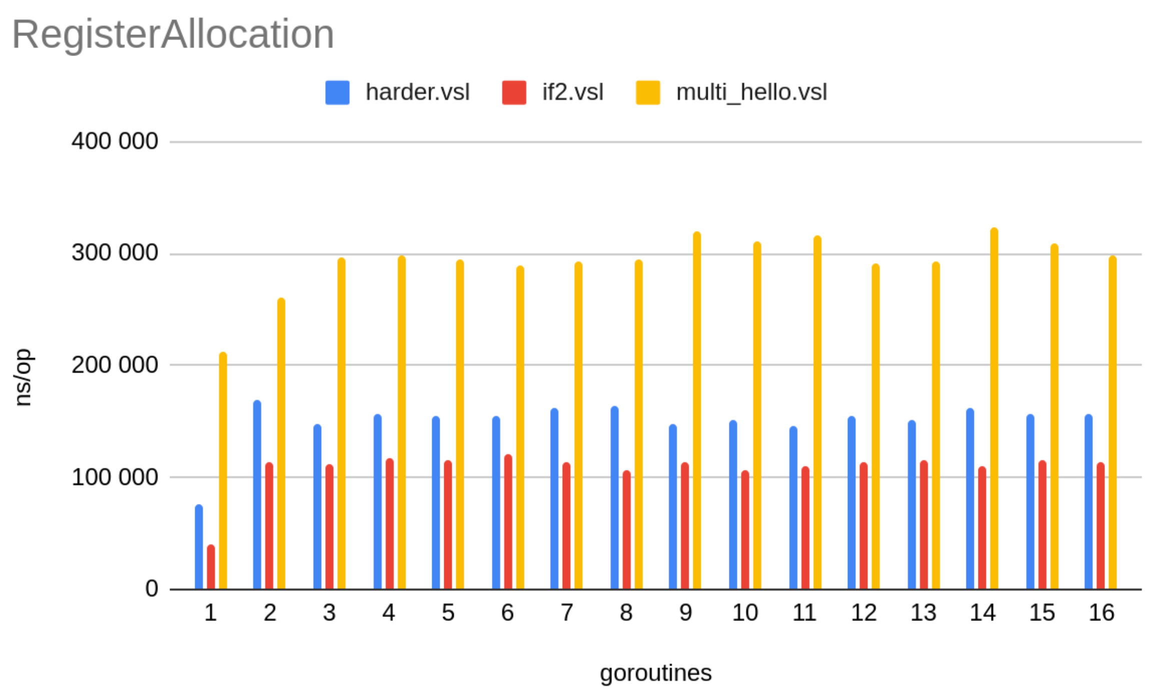


Figure 15: The figure shows that smaller file sizes don't benefit from many goroutines, even though the task is easily parallelisable. Lower ns/op is better.

In stark contrast to the considerable speedup of the *aamanyfuncs.vsl* source file, the remaining three source files experience the opposite effect. They slow down. This is somewhat unexpected, as the task is highly parallelisable. However, this might be because the files are relatively small, and as such there might be costs of the initial allocation of goroutines and subsequent LIR function object allocation.

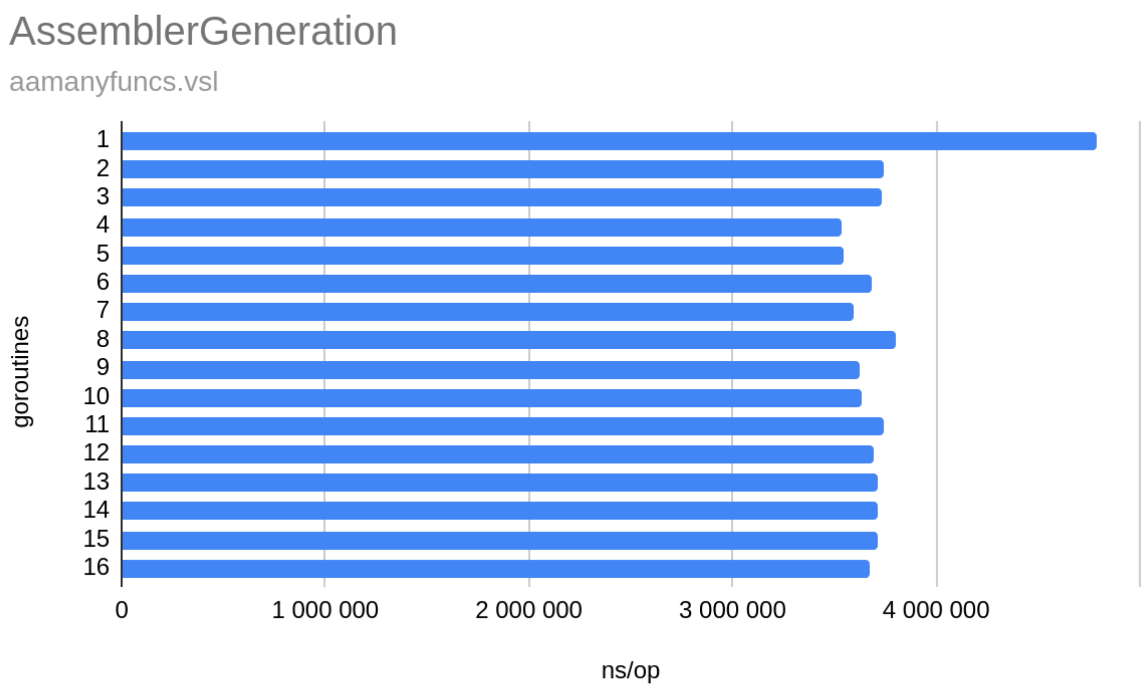


Figure 16: The figure illustrates the general trend for the large source file *aamanyfuncs.vsl* as was seen for the Aarch64 benchmark..There is an immediate performance increase with two goroutines, but as many more goroutines are introduced, there are diminishing returns. Lower ns/op is better.

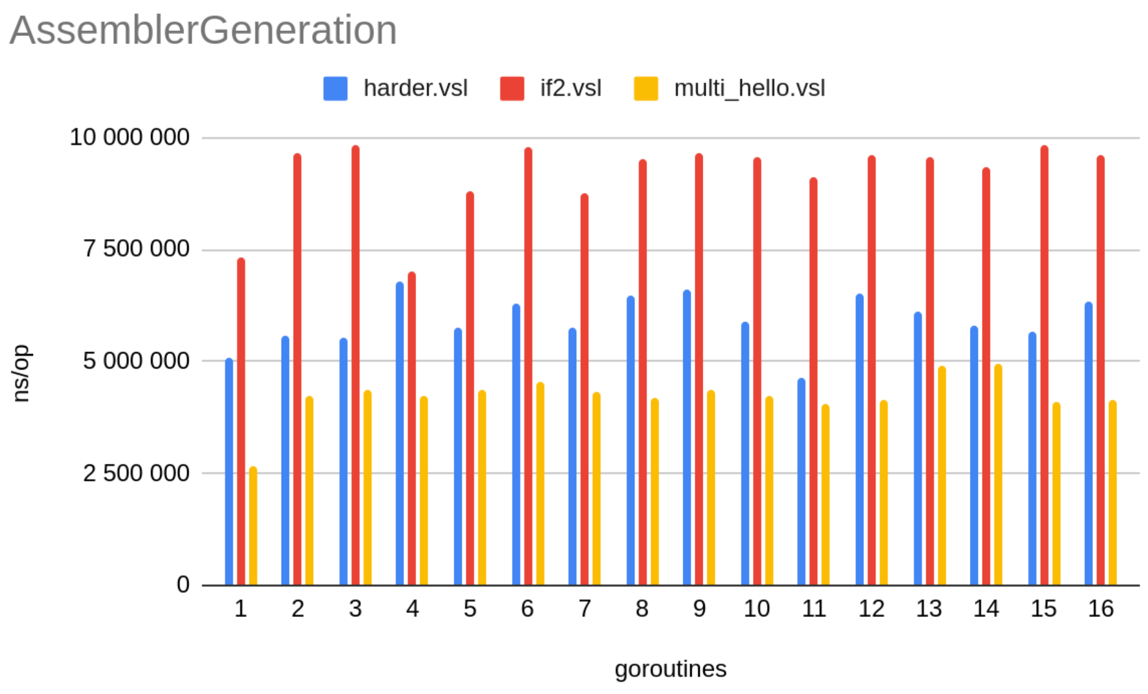


Figure 17: Parallel assembler generation provides no speedup for the smaller file sizes. On the contrary, they slow the process down. Lower ns/op is better.

Lastly we look at outputting the assembler code from the LIR SSA with assigned hardware registers. The large *aamanyfuncs.vsl* source file experiences an immediate 28.1% performance improvement at two goroutines. The improvement quickly stagnates, as more goroutines are introduced.

The smaller file size source files follow the same trends we have seen previously. They slow down when function parallelism is introduced. *if2.vsl* stands out as the slowest to process for the first time, because of the multiple basic block generations due to single line if-code fragments resulting in sizable assembler output. As ever, the introduction of multiple worker threads slows things down, compared to the sequential compiler.

The final observation is that the most of the collected work during this particular compilation process are AST optimisation and writing the assembler to file. For the sequential compiler, register allocation was a particularly strenuous process for the large *aamanyfuncs.vsl* source file.

4.3 Compiler source code size

The subsequent tables show line counts for both the TDT4205 2021 edition VSL compiler written in C and the typed VSL-compiler written in Go. Comment lines are lines that *begin* with “//” for Go and “//” or “/*” for C respectively. Files with other file extensions than .go, .h or .c were not counted.

	C-source code lines	**C with Flex and Bison generated C-files**
Source code lines	1882	7856
Comments	138	290

Table 7: Code line count for the C-equivalent TDT4205 2021 course.

The written C source code line count is relatively small compared to the Flex and Bison generated source code. The total code size in lines increases by 417% when the state machine and parser is generated.

	Go source code lines
Source code lines	10626
Comments	1106

Table 8: Code line count for the parallelisable compiler. Total source code size includes the benchmarking source file.

The Go source code is much larger than the C equivalent, even taking into consideration the Flex and Bison generated code. However, the compiler has more capability in that it is parallelisable and interoperates with LLVM. The below table shows the number of lines of code per Go package.

package	frontend	ir	ir/llvm	ir/lir	backend	util
Lines	1640	5952	1364	3954	1780	736

Table 9: Source code lines per package of the parallelisable compiler written in Go.

The *ir/lir* package dominates the line counts of the *ir* package. 913 lines belong to the *ir/lir/transform.go* AST-to-LIR transformer. The remaining 3,041 lines could potentially be extracted into an external package. This solution would be similar to the LLVM source code found in *ir/llvm/transform.go*. Such a separation represents a 28.6% reduction in overall source code size and a 23.1% reduction in code size of the *ir/lir* package. This separation would leave the compiler at a total line count of 7,585, which is just smaller than the C-equivalent with the generated scanner and parser.

If we were to leave out parallel code from the compiler we could save additional line counts for a potential sequential replacement compiler. Alterations, deleting parallel code lines, to the following files will accumulate in several hundreds fewer lines of code.

- Remove *util/perror.go*, saving 76 lines of code. It is only used by the parallel code.
- Remove *util/args.go* and use the flag package built into Go [43], saving 195 lines of code.
- Remove 90 lines of code from *ir/lir/transform.go*.
- Remove 98 lines of code from *ir/llvm/transform.go*.
- Remove 36 lines of code from *ir/optimise.go*.
- Remove 34 lines of code from *backend/lir/regalloc.go*.
- Remove 29 lines of code from *backend/arm/armv8.go*.

package	frontend	ir	ir/llvm	ir/lir	backend	util	Total
Parallel	1,640	5,952	1,364	3,954	1,780	736	10,108
Sequential	1,640	5,738	1,266	3,864	1,717	465	9,560

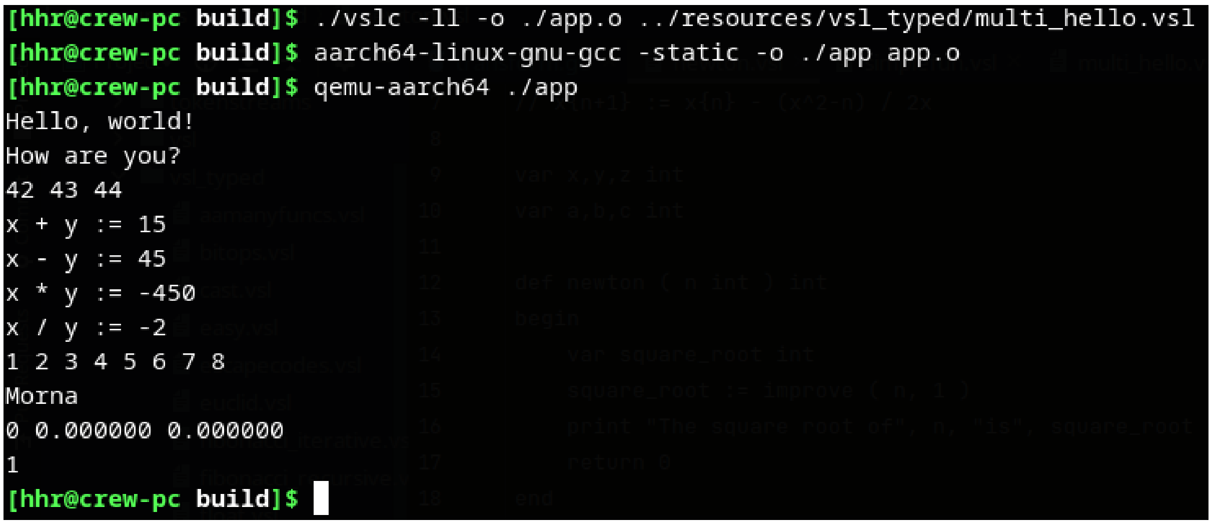
Table 10: Source code lines per package of parallelisable compiler written in Go, without parallel code, or parallel code dependencies. All line numbers are excluding in-code comments.

If we again subtract the separable *ir/lir* code from both the parallel and sequential compiler source codes, we get 10,108 - 3,041 = 7,067 and 9,560 - 3,041 = 6,519 lines of code respectively. The delta is 7,067 - 6,519 = 548 lines of code, which represents a 7.8% code reduction. Comparing this new sequential Go compiler with the previous C, Flex and Bison compiler yields 7,856 - 6,519 = 1,337 fewer lines of code, a reduction of 17.0%.

4.4 LLVM integration

Compiler-to-LLVM integration was successful using the *tinygo/go-llvm* module [44]. Another successful, intermediate frontend-to-LLVM IR transpiler was created, and later removed, using the *llir/llvm* module [45]. The latter was removed because it does not interoperate with the LLVM toolchain, but rather conforms to the LLVM standard.

The former module enables the parallelisable compiler, using the *-ll* flag, to generate target specific machine code using the C-bindings and the system installed LLVM toolchain. The transformation of AST-to-LLVM works in similar fashion to AST-to-LIR, found in the *src/ir/llvm/transform.go* compiler source file. The target code is an object file, which must be linked. An example is given below.



```
[hhr@crew-pc build]$ ./vslc -ll -o ./app.o ../resources/vsl_typed/multi_hello.vsl
[hhr@crew-pc build]$ aarch64-linux-gnu-gcc -static -o ./app app.o
[hhr@crew-pc build]$ qemu-aarch64 ./app
Hello, world!
How are you?
42 43 44
x + y := 15
x - y := 45
x * y := -450
x / y := -2
1 2 3 4 5 6 7 8
Morna
0 0.000000 0.000000
1
[hhr@crew-pc build]$
```

Figure 18: This figure illustrates the usage of the vslc compiler and the *-ll* flag to invoke the system's installed LLVM toolchain to generate target *object code*.

The above figure illustrates the compilation process. The source file is compiled into an *object file*, which is linked with the aarch64-specific gcc compiler to create the final aarch64 executable.

5 Discussion

5.1 The parallelisable compiler

This chapter is dedicated to discuss the findings of chapter 4.2. With the relevant data sets, relative execution time ratios and illustrative figures in mind, I present each major compiler stage, or process, in the subsequent sub-chapters.

5.1.1 Frontend

I admit to the fact that this chapter, parallel frontend, does not address any data driven benchmark, contrary to the subsequent two chapters. Nevertheless, I believe it is an interesting topic for discussion. I've decided to include it because the topic *is relevant* for a parallelisable compiler.

The compiler accesses a file by moving the file's content into memory. Once the file contents have been stored in memory, the compiler will begin scanning the file for lexemes. This lexing process is intuitively a sequential process. The reason being that a lexer, with respect to function level parallelism, cannot know where a function begins and ends, based on the sequential lexemes it generates for the parser. A lexer only knows how far into a sequence of bytes it has scanned, and where the current lexeme starts.

A possible workaround solution for this issue is presented by Gerhards and Lonvick in their standardisation of a common computer network logging protocol [17]. This syslog protocol suffers from a similar problem that a compiler lexer does: *if there are multiple entries, where is the index which separates any two entries in the message*? A proposed, and viable solution, is the use of octet counting [17]. Every syslog message is prepended by a digit that states the length of the following entry in bytes, effectively telling the syslog parser where the subsequent entry starts. An example is given below.

The two syslog messages are sent as two distinct transmissions to a syslog receiver.

```
<34>Oct 10 2000 12:34:00 client01 my harddisk is nearly full
<165>Oct 10 2000 12:34:00 client01 harddisk free space is 1.7GB
```

With octet counting enabled, the messages can be sent in one transmission to the receiver. The prepended digit, 60, tells the parser how long the initial message is, in bytes, and *implicitly*; where the subsequent message, carried by this single transmission, begins. Do note that the newline character is not included in the byte sequence constituting the syslog message transmission, but is appended here for readability.

```
60 <34>Oct 10 2000 12:34:00 client01 my harddisk is nearly full
<165>Oct 10 2000 12:34:00 client01 harddisk free space is 1.7GB
```

This method should be applicable to any text based data, like syslog messages or program source code, that has predefined structure. However, where the former is a system generated text message, the latter is human made. This distinction is essential to explaining why octet counting isn't viable for function level parallelism in a compiler setting. Assume that octet counting is prepended for every function in a source code file. If only a single byte that constitutes the function source code is removed or added, the programmer, or the editing software, has to edit the prepended octet count. Additionally the programmer has to prepend the correct amount of bytes, if done manually. The potential for human errors is considerable, and renders this solution infeasible for source code editing without explicit automatic editing software support.

Additionally, this parallel scanning would require some parallel parser that constructs a single syntax tree from multiple individual parse trees, while retaining program order.

5.1.2 AST transformations

Parallelisation of the AST optimisation and LIR generation passes were underwhelming, to say the least. The *aamanyfuncs.vsl* source file, which constitutes a relatively large tree structure, compared to the remaining three source files, should provide a great basis for parallel tree traversal. The results, on the other hand, report an increase in compilation time (AST optimisation) or a negligible decrease (LIR generation).

Tree traversal algorithms are categorised as irregular. They exert unpredictable memory accesses, because the sequence of nodes in the tree visited may differ based on the tree's nodes and on the state of the traversee [47]. Goldfarb, Jo and Kulkarni suggest that "(...), after traversing one path in the tree, a thread must return to higher levels of the tree to traverse down other children (...). As a result, interior nodes of the tree are repeatedly visited during traversals, requiring in substantial extra work" [47]. Transforming the tree structure, by deleting or rearranging nodes or sub-trees, increases complexity and subverts the effects of parallelism [46].

An alternative explanation of the AST optimisation results may be that the benchmark parses the AST from the source string on every iteration. This has to be done because the AST optimisation pass mutates/transforms the AST. It may be that the parsing process is so expensive to the point where the AST optimisation process' execution time or memory allocations are negligible by comparison.

Generating the LIR IR from the AST did not experience slowdown akin to the AST transformation. The expected speedup still eludes the writer, save for a 31.6% performance increase with two goroutines from one. Subsequent performance levels are nearly constants for greater numbers of goroutines. As is reported by Marlow and Jones, what appears initially parallelisable in theory, is in practice less so, with the introduction of global dependencies [49]. In the vslc compiler, the module and mutexes represent global state.

The three smaller files, *harder.vsl*, *if2.vsl* and *multi_hello.vsl* might not even be sufficiently large to provoke any opportunity to exploit parallelism. What little speedup is possible vanishes in the diminishing returns of parallel setup and the aforementioned mutexes and global LIR module.

5.1.3 Target generation

During assembler code generation the need for translating LIR virtual registers into hardware registers of the target architecture arises. This raises two problems. How to translate an infinite amount of virtual registers into *finite* hardware registers, and what happens when we can no longer allocate available hardware registers.

I solved the first problem, *register allocation,* using the map colouring algorithm [42]. The map colouring algorithm attempts to assign registers based on variable dependencies and interference graphs. The idea is to minimise the number of registers needed, such that we can fit as many variables in registers as possible. Fegaras states that "If two variables do not interfere (...) then we can use the same register for both of them, thus reducing the number of registers needed" [10].

The second problem, *register saturation*, can be solved by register spilling. When no more registers can be assigned the register lives in memory, and must be loaded and stored accordingly [28]. Because neither VSL source file defines code that saturates the available registers of the target architecture, I decided to not implement register spilling in favour of spending time and effort elsewhere. If the compiler detects register saturation it will return an error stating that it cannot compile the program because register spilling is not implemented.

The register allocation pass was by far the most successful with regards parallelisation. The *aamanyfuncs.vsl* file shows a compilation time diagram of what we'd expect from easily parallelisable, structured data. This is achievable because there are no global dependencies or mutexes, the data is arranged linearly using a two-dimensional array of basic blocks and instructions per basic block, effectively a matrix-like structure.

The writing of the assembler to file follows the same trend shown by LIR generation. However, where LIR generation may have been subverted by the tree structure of the AST, the generation of assembler must experience degrading improvements elsewhere, as the source format is already LIR, a two-dimensional matrix-like structure. Pondering these results led me to profile two benchmarks in the benchmark suite, namely *BenchmarkLIRGeneration* and *BenchmarkAssemblerGeneration*. The profiling data can be viewed in Appendix E and Appendix F, for CPU and memory respectively. Results are also available in pprof profile formats in Appendix B in the */doc/profiles* directory [55]. Profiles were conducted for both benchmarks, with the parallelism of the vslc compiler set to *-t* equals one, two, four and 16 worker goroutines.

By first looking at the CPU profiling data we see that most CPU time (flat) is spent in runtime or memory memory bound operations (printf). *runtime.mallocgc* ranks in the top five CPU usage times (flat) for ⅞ profiling data sets, where LIR generation with -t = 2 is the odd one out. *mallocgc* is the malloc-equivalent for Go, and is invoked when allocating memory to data such as arrays, structs, maps, etc. [51]. *runtime.scanobject* also ranks high among many profiles. *scanobject* scans heap memory for data objects given an address [50]. The common denominator is the go runtime garbage collector (GC). The GC is scheduled to run when heap allocations cause the heap to grow beyond the GOGC variable, which by default means when the heap doubles in size [52]. When this limit is reached, the GC is scheduled to perform removal of no longer used memory allocations. Time spent on mutexes did not dominate the parallel compilation runs as much as I had initially thought (*runtime.futex*). The difference in mutex CPU time is higher in the assembler generation profiles than in LIR generation profiles. This suggests that LIR generation and assembler generation define memory bound routines.

Looking at the memory profiles for *LIR generation* we see that approximately 70% of all memory allocations are caused by *lir.(*Block).CreateLoad* and *lir.(*Block).CreateStore*. This is most likely caused by *aamanyfuncs.vsl's* numerous load and stores from declared variables. The function's themselves only allocate a single instruction data object. However, this is not a helpful observation, as the sequential compiler only allocates at most ¾ as much memory as any parallel run. In general, memory allocations for this pass are dominated by the nature of the program defined by source file, not the compiler parallelism. I conclude that LIR generation suffers from the aforementioned subversion of parallelism for tree structures, settling on the conclusion of chapter 5.1.2.

The *assembler generation* memory profiles show extensive use of the *strings.(*Builder).WriteString* method of the *strings.Builder* struct (hereafter referred to as *builder*). The *builder* is a memory buffer (slice of bytes) that is appended by every call to *WriteString*. When the buffer can't accommodate a new string, the buffer is expanded using the slice *append* function [53]. Appending a slice causes allocation of a new slice, with either double size or some requested size. The contents of the old slice are copied into the new slice [54]. The buffer is uninitialised at declaration, meaning it has to be allocated some time later. This is a major performance factor, because the builder's buffer holds the output string data. For the *sequential* run, *one single* builder is allocated and used for buffering all functions. In the *parallel* run, one builder *per worker goroutine* is assigned for holding their respective buffered assembler code. This causes the parallel compiler to declare multiple uninitialised buffers, that have to be grown, at run time when needed.

If ***m*** denotes the number of *append* calls on the sequential compiler's buffer, ***t*** is the number of worker go routines for the parallel compiler and n_i denotes number of times *append* calls for each worker thread i, I expect the following to be generally true for source files with many functions:

$$m \leq \sum_{i=2}^{t} n_i, \; t \geq 2$$

This claim is founded by the fact that, if we can expect append to double the size of the underlying slice, then, at some point, the growth of the sequential compiler's append will accommodate the remaining functions faster than the total growth required by multiple appends for parallel worker go routines with individual buffers. Stated differently, the more append is called on the same slice, the greater the increase in the slice's capacity. Appendix G shows an example of buffer allocation using append for two data types (int and byte array), while Appendix H proves the statement.

A possible mitigation to the above problem would be to heuristically pre-allocate buffer space for the builders, using the *string.(*Builder).Grow* method. Pre-allocation can be done on a function level basis, because we know that a function must at least set up a minimal stack for SP and LR, any declared parameters, and local variables. It's also possible to pre-allocate based on the number of instructions carried by function, by iterating over its basic blocks, multiplying the sum with some pre-calculated average of all instructions length in bytes.

5.1.5 Output

Having attested that a VSL program can be compiled to assembler in parallel, and produce the *equivalent* result, I want to point out one particular detail that may affect the end result after assembling *and* linking. Given two example functions in the same VSL file, and assume an alteration to the VSL specification allowing the main-function to call both functions *foo* and *bar* from main.

```
def foo() int
begin
	var b int
	b := bar(2)
	print "b is ", b
	return 0
end

def bar(b int) int
	return b * 2
```

Code snippet 5: A sample VSL program. The main function is created implicitly by the vslc compiler, as Bennet's definition of VSL calls the first function defined in the source file.

In a sequential run of the vslc compiler this code will compile into the imaginary assembler of the form given in the below code snippet *Run A*. During parallel compilation however, there is no guarantee that function *foo* will be written to the output stream as assembly code before function *bar*. Parallel compilation may result in assembly code of *Run B*. Every instruction has length equal to one machine word.

Run A (also sequential)	**Run B**
`foo:`	`bar:`
`        add    r0, r0, r1`	`        load   r0, [sp, #8]`
`        ret`	`        sub    r0, #16`
	`        store  r0, [sp, #0]`
`bar:`	`        mul    r0, r0, #5`
`        load   r0, [sp, #8]`	`        store  r0, [sp, #8]`
`        sub    r0, #16`	`        mov    r0, #0`
`        store  r0, [sp, #0]`	`        ret`
`        mul    r0, r0, #5`	
`        store  r0, [sp, #8]`	`foo:`
`        mov    r0, #0`	`        add    r0, r0, r1`
`        ret`	`        ret`
`main:`	`main:`
`        call   foo`	`        call   foo`
`        call   bar`	`        call   bar`
`        ret`	`        ret`

Code snippet 6: Run A is the imaginary sequential, and possible parallel, assembler of running the vslc compiler on the source code of Code snippet 5. Run B is an alternate result when running vslc in parallel mode, where scheduling causes functions foo and bar to be written to the output file in different order to the sequential.

Codes *Run A* and *Run B* define equivalent programs, but they differ in that functions *foo* and *bar*, obviously, have swapped order. The observation that *Run A* and *Run B* are equivalent is true for assembler language, because the subsequent process of assembling and linking the assembler file will generate jump offsets based on code length in bytes, not by labels. It's easy to conclude that parallel output to an executable binary cannot be achieved the way this compiler is constructed.

In code snippet *Run A*, the offset from *main's* call to *foo* will be nine machine words (the length of function *foo's* and *bar's* instructions), while the offset to *bar* will be eight machine words (the length of function *bar's* instructions). Code snippet *Run B* shows the equivalent program where *foo* and *bar* have had their positions in the output assembler swapped, because scheduling caused the goroutine generating *bar* to complete before the goroutine generating *foo*. If the same offsets from the function calls of *main* are retained, nine and eight words for *foo* and *bar* respectively, the call to *foo* will result in calling *bar*, while calling *bar* will result in setting the program counter to the second instruction of the function *bar*.

5.2 Go as a compiler implementation language

In this chapter I want to cover some less data driven discussion on compiler implementation, specifically with the Go programming language. The subsequent sections will emphasise some general benefits, weaknesses and observations using the Go programming language in creating both a parallelisable compiler and an elementary compiler for students to implement.

One of the great disadvantages of Go, and many non-object oriented programming languages, is revealed when constructing the LIR intermediate representation library. All LIR instructions must conform to the *Value interface*. An interface is a contract that a type, in Go's instance, must conform to by implementing the interface's methods [33]. A structure must implement the interface's functions as methods on itself. In the example of LIR, all instructions must implement the Name(), Id() and DataType()-methods to name a few. Whenever a new type of instruction is created, such as a struct type, the interface functions must be created for the newly created struct to conform to the Value interface. This generates a lot of duplicate code. Object oriented programming languages can mitigate this greatly by exploiting inheritance, polymorphism and method overloading. Inheritance enables subclasses to specialise the properties of the superclass, while the superclass generalises the properties of its subclasses [18]. This enables the programmer to define, let's say a String() method on the superclass, and overload the String() method in a subclass, *if it's needed*.

The goyacc library has very limited error reporting. Whenever an error occurs, the parse function simply returns 1, while the informative error message is passed to the lexer, which effectively violates Go error passing, by propagating the error to the callee, not the caller. Error messages can be enabled by explicitly editing the goyacc generated Go source code.

Chapter 4.3 showed that the parallelisable compiler has fewer lines of code than the 2021 edition TDT4205 C, Flex and Bison equivalent, if the LIR package was separated into a separate source module. If the parallelisable compiler was to be used as the basis for a new TDT4205 compiler, it would be reasonable to prune the implementation of any parallel code. Pruning parallel code results in 7.8% code reduction, measured in lines of code, and a 17.0% reduction compared to the C, Flex and Bison equivalent.

6 Related work

The ParallelGcc project is “(...) a research project aiming to parallelize a real-world compiler”, namely the gcc compiler toolchain [35]. ParallelGcc parallelises the compiler optimisation passes based on *Intra Procedural Analysis*, which “are optimizations which are applied inside a function, ignoring its call and callee relationship” [35]. This definition is similar to function level dependency, introduced in this thesis. In its current version, the ParallelGcc compiler has parallelised the GIMPLE *Intra Procedural Analysis* optimisation passes only. Benchmarking the *gimple-match.c* source file they managed a 72% speedup of the GIMPLE Intra Procedural Analysis optimisation pass, and a 6% total gcc speedup going from one thread to two. Four threads provided speedups of 152% and 9% for GIMPLE Intra Procedural Analysis and total gcc process respectively. They estimate a total gcc speedup of 35% and 61% when RTL parallelisation is complete, for two and four threads respectively. The parallelisation of the RTL is in the rota for the project.

Dr. Aaron Wen-yao Hsu explored how tree structures can be transformed by a parallel process [46]. Hsu argues that “(...) the pointer-based top-down record-type representation (...)” commonly used in many tree structures today demonstrate inferior performance to array-oriented programming and traditional APL programming [46]. Additionally Hsu notes that conditionals can be discarded with this array-oriented paradigm using traditional array operations. He also notes that traditional trees based on pointers and record-types are difficult to parallelise without detailed knowledge of the tree or making these transformations application specific, because trees often exhibit irregular structure. Hsu adds that transforming these traditional tree structures are memory bound processes, not compute bound.

7 Conclusion

The parallelisable compiler has produced two interesting observations of function level parallelism. Source files with multiple *large* functions benefit from being compiled in parallel with respect to function level parallelism. The AST optimisation pass is the only process where increased parallelism doesn't improve compiler performance. However no conclusion can be drawn whether the unexpected result is a result of subverted parallelism because of memory bound operations or inadequate benchmark routine. Register allocation, on the other hand, displays considerable speedup. *Smaller* source files, with few functions or a single function, don't benefit from function level parallelism. Quite the opposite, the compilation time increases when going from a sequential compiler to a parallel compiler.

Parallelising the frontend and backend output seems problematic. The first is problematic because scanning lexemes is by nature a sequential process. Introducing octet counting would make it parallelisable, but would require source code editing software support. The backend faces another obstacle, that of program order. Because the programmer cannot control the scheduler that issues threads/goroutines, they cannot control which output is written to the output file in which order, causing undefined behaviour with jump instructions being misaligned.

The Go programming language appears to be a capable compiler implementation language that offers much of the same functionality that the C programming language paired with the Flex and Bison softwares do. Highlights are built-in string data type, performant built-in hashmap, minimal code footprint and good operating system independence. Downsides of the Go programming language is, from the perspective of the LIR SSA, no object orientation and inheritance. The lack of inheritance makes interfaced types tedious to implement when multiple types have to perform the same operations.

Further work could look into how one might create a parser that is independent of the goyacc parser generator. The official LLVM tutorial, constructing the simple mathematical language Kaleidoscope, utilises recursive descent and operator-precedence parsers that accept a token stream to build an AST [32]. This represents a great increase in code length however, as opposed to the Bison grammar of the current TDT4205 course [16, 32]. An interesting experiment would be to measure the total compiler performance with the AST optimisation pass removed. If the LIR IR can be generated from the un-optimised AST, the compiler can cut time by discarding the most time consuming compiler pass. Additionally it is desirable to lower the memory allocations of the assembler generation pass. This can be partly achieved by pre-allocating the output buffers using heuristics and meta information easily accessible from the LIR IR.

References

[1] M. Goldman and R. C. Miller, “Reading 17: Concurrency,” web.mit.edu, 2014. https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/ (accessed Feb. 20, 2022).

[2] T. B. Skaali, “Real Time and Embedded Data Systems and Computing Concurrency and concurrent systems,” 2011. Accessed: Feb. 20, 2022. [Online]. Available: https://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/forelesninger-rt/2011-2_Concurrent_systems.pdf

[3] A. Grama, G. Karypis, V. Kumar, and A. Gupta, Introduction to parallel computing, 2nd edition. Harlow, England ; New York: Addison-Wesley, 2003.

[4] G. S. Almasi and A. Gottlieb, Highly parallel computing. Redwood City Benjamin Cummings, 1989.

[5] R. Pike, “Concurrency is not Parallelism,” talks.golang.org, Jan. 11, 2012. https://talks.golang.org/2012/waza.slide#3 (accessed Feb. 21, 2022).

[6] Heroku and R. Pike, “Concurrency Is Not Parallelism,” Vimeo, Sep. 18, 2012. https://vimeo.com/49718712 (accessed Feb. 21, 2022).

[7] B. A. Schreiber et al., “function | Definition, Types, Examples, & Facts,” Britannica. 2019. Accessed: Feb. 21, 2022. [Online]. Available: https://www.britannica.com/science/function-mathematics

[8] K. E. Aubert and H. Holden, “funksjon – matematikk,” Store norske leksikon, Jul. 21, 2021. https://snl.no/funksjon_-_matematikk (accessed Feb. 21, 2022).

[9] Apple Swift, “Functions — The Swift Programming Language (Swift 5.6),” docs.swift.org, Jan. 27, 2022. https://docs.swift.org/swift-book/LanguageGuide/Functions.html (accessed Feb. 21, 2022).

[10] L. Fegaras, “CSE 5317/4305: Design and Construction of Compilers,” Uta.edu, Jan. 20, 2015. https://lambda.uta.edu/cse5317/notes/short.html (accessed Nov. 30, 2021).

[11] J. P. Bennett, Introduction to compiling techniques : a first course using ANSI C, Lex and YACC, 2nd edition. London: Mcgraw-Hill, 1996.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms, 3rd edition. Cambridge, Massachusetts: Mit Press, 2009, pp. 253–285.

[13] D. Cheney, “How the Go runtime implements maps efficiently (without generics) | Dave Cheney,” Dave Cheney, May 29, 2018. https://dave.cheney.net/2018/05/29/how-the-go-runtime-implements-maps-efficiently-without-generics (accessed Feb. 22, 2022).

[14] R. Pike, “Lexical Scanning in Go - Rob Pike,” www.youtube.com, Aug. 30, 2011. https://www.youtube.com/watch?v=HxaD_trXwRE (accessed Nov. 30, 2021).

[15] K. D. Cooper, K. Kennedy, and L. Torczon, “Lexical Analysis - An Introduction,” University of Massachusetts, 2003. https://people.cs.umass.edu/~moss/610-slides/04.pdf (accessed Feb. 23, 2022).

[16] M. Engel, “TDT4205,” folk.ntnu.no, Jan. 11, 2021. https://folk.ntnu.no/michaeng/tdt4205_21/ (accessed Nov. 24, 2021).

[17] R. Gerhards and C. Lonvick, “RFC 6587 - Transmission of Syslog Messages over TCP,” datatracker.ietf.org, Apr. 2012. https://datatracker.ietf.org/doc/html/rfc6587 (accessed Apr. 08, 2022).

[18] P. Wegner, “Concepts and paradigms of object-oriented programming,” ACM SIGPLAN OOPS Messenger, vol. 1, no. 1, pp. 7–87, Aug. 1990, doi: 10.1145/382192.383004.

[19] D. Vyukov, “Scalable Go Scheduler Design Doc,” Google Docs, May 02, 2012. https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit (accessed May 13, 2022).

[20] A. Freeman, PRO GO : the complete guide to programming reliable and efficient software. S.L.: Apress, 2022.

[21] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” Journal of the ACM, vol. 46, no. 5, pp. 720–748, Sep. 1999, doi: 10.1145/324133.324234.

[22] N. Deshpande, E. Sponsler, and N. Weiss, “Analysis of the Go runtime scheduler,” 2012.

[23] Go Authors, “- The Go Programming Language,” go.dev, 2014. https://go.dev/src/runtime/proc.go (accessed May 13, 2022).

[24] B. Eastwood, “The 10 Most Popular Programming Languages to Learn in 2020,” Northeastern University Graduate Programs, Jun. 18, 2020. https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/ (accessed May 13, 2022).

[25] A. Gerrand, “Go version 1 is released - The Go Programming Language,” go.dev, Mar. 28, 2012. https://go.dev/blog/go1 (accessed May 13, 2022).

[26] TIOBE, “TIOBE Index | TIOBE - The Software Quality Company,” Tiobe.com, May 2022. https://www.tiobe.com/tiobe-index/ (accessed May 13, 2022).

[27] Stack Overflow, “Stack Overflow Developer Survey 2021,” Stack Overflow, 2021. https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies (accessed May 13, 2022).

[28] M. Braun and S. Hack, "Register Spilling and Live-Range Splitting for SSA-Form Programs," in LNCS 5501, York, United Kingdom, 2009, pp. 174–175. Accessed: May 13, 2022. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-642-00722-4.pdf

[29] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88, 1988, doi: 10.1145/73560.73561.

[30] Intel Ark, "Product Specifications," www.intel.com, 2010. https://ark.intel.com/content/www/us/en/ark/products/47924/intel-xeon-processor-e5630-12m-cache-2-53-ghz-5-86-gts-intel-qpi.html (accessed May 15, 2022).

[31] Intel Ark, "Product Specifications," www.intel.com, 2012. https://ark.intel.com/content/www/us/en/ark/products/64896/intel-core-i53320m-processor-3m-cache-up-to-3-30-ghz.html (accessed May 15, 2022).

[32] LLVM Foundation, "2. Kaleidoscope: Implementing a Parser and AST — LLVM 6 documentation," releases.llvm.org. https://releases.llvm.org/6.0.0/docs/tutorial/LangImpl02.html (accessed May 20, 2022).

[33] golang, "A Tour of Go," go.dev. https://go.dev/tour/methods/9 (accessed May 20, 2022).

[34] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach. Cambridge, Ma: Morgan Kaufmann, 2019.

[35] G. Belinassi, "ParallelGcc - GCC Wiki," Gnu.org, Dec. 16, 2019. https://gcc.gnu.org/wiki/ParallelGcc (accessed May 23, 2022).

[36] P. L. & S. Burke, "New CPU Test Methodology 2020: Code Compile, Updated Gaming, Transcoding, & More," www.gamersnexus.net, May 02, 2020. https://www.gamersnexus.net/guides/3577-cpu-test-methodology-unveil-for-2020-compile-gaming-more (accessed May 23, 2022).

[37] "Why the Build System is Slow — Firefox Source Docs documentation," firefox-source-docs.mozilla.org. https://firefox-source-docs.mozilla.org/build/buildsystem/slow.html (accessed May 23, 2022).

[38] Kernel build community, "Building Linux with Clang/LLVM — The Linux Kernel documentation," docs.kernel.org. https://docs.kernel.org/kbuild/llvm.html (accessed May 23, 2022).

[39] Alibaba Tech, "GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers," Medium, Dec. 27, 2019. https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378 (accessed May 23, 2022).

[40] M. Wolfe, “Parallelizing compilers,” ACM Computing Surveys, vol. 28, no. 1, pp. 261–262, Mar. 1996, doi: 10.1145/234313.234417.

[41] LLVM Foundation, “LLVM Reference Manual,” llvm.org, May 23, 2022. https://llvm.org/docs/LangRef.html#type-system (accessed May 23, 2022).

[42] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring,” citeseerx.ist.psu.edu, 1981. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.452.8606&rep=rep1&type=pdf (accessed May 24, 2022).

[43] golang authors, “flag package - flag - pkg.go.dev,” pkg.go.dev, May 10, 2022. https://pkg.go.dev/flag (accessed May 24, 2022).

[44] A. van Laëthem, E. Sales de Andrade, F. G. Schwindt, J. Clift, and 美迫, “Go bindings to system LLVM,” GitHub, May 23, 2022. https://github.com/tinygo-org/go-llvm (accessed May 26, 2022).

[45] R. Eklind, L. Tsú-thuàn, P. Waller, A. Ballholm, S. Ser, and J. Clift, “llvm,” GitHub, May 25, 2022. https://github.com/llir/llvm (accessed May 26, 2022).

[46] A. Wen-yao Hsu, “A DATA PARALLEL COMPILER HOSTED ON THE GPU,” PhD/MSc, Indiana University, 2019. Accessed: May 27, 2022. [Online]. Available: https://scholarworks.iu.edu/dspace/bitstream/handle/2022/24749/Hsu%20Dissertation.pdf?sequence=1&isAllowed=y

[47] M. Goldfarb, Y. Jo, and M. Kulkarni, “General transformations for GPU execution of tree traversals,” Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Nov. 2013, doi: 10.1145/2503210.2503223.

[48] golang, “goyacc command - golang.org/x/tools/cmd/goyacc - pkg.go.dev,” pkg.go.dev, Mar. 15, 2022. https://pkg.go.dev/golang.org/x/tools/cmd/goyacc (accessed May 29, 2022).

[49] S. Marlow and S. Peyton Jones, “Download Limit Exceeded,” citeseerx.ist.psu.edu, Mar. 16, 2012. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.378.6463&rep=rep1&type=pdf (accessed May 30, 2022).

[50] golang, “Source file src/runtime/mgcmark.go,” go.dev, 2009. https://go.dev/src/runtime/mgcmark.go (accessed May 31, 2022).

[51] golang, “Source file src/runtime/malloc.go,” go.dev, 2014. https://go.dev/src/runtime/malloc.go (accessed May 31, 2022).

[52] D. Cheney, “GOGC | Dave Cheney,” cheney.net, Nov. 29, 2015. https://dave.cheney.net/tag/gogc (accessed May 31, 2022).

[53] golang, “Source file src/strings/builder.go,” go.dev, 2017. https://go.dev/src/strings/builder.go (accessed May 31, 2022).

[54] golang, “src/runtime/slice.go,” GitHub, Sep. 25, 2020. https://github.com/golang/go/blob/go1.16.7/src/runtime/slice.go (accessed May 31, 2022).

[55] golang, “pprof package - runtime/pprof - pkg.go.dev,” pkg.go.dev, May 10, 2022. https://pkg.go.dev/runtime/pprof (accessed May 31, 2022).

[56] golang, “testing package - testing - pkg.go.dev,” pkg.go.dev, Jun. 01, 2022. https://pkg.go.dev/testing (accessed Jun. 02, 2022).

[57] D. Cheney, “How to write benchmarks in Go | Dave Cheney,” cheney.net, Jun. 30, 2013. https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go (accessed Jun. 02, 2022).

[58] M. Kerrisk, “time(1) - Linux manual page,” man7.org, Mar. 06, 2019. https://man7.org/linux/man-pages/man1/time.1.html (accessed Jun. 02, 2022).

Appendices

Appendix A: Hashmap comparison source code Git repository

Hash comparison source code: https://github.com/hhramberg/hashComparison

Appendix B: Compiler source code Git repository

Typed VSL compiler source code: https://github.com/hhramberg/go-vslc

Appendix C: Benchmark results parallel compiler raw

```
Benchmark, Filename, #goroutines, n, Result
Aarch64, aamanyfuncs.vsl, 1, 56, 21012871
Aarch64, aamanyfuncs.vsl, 2, 67, 17224975
Aarch64, aamanyfuncs.vsl, 3, 68, 16913841
Aarch64, aamanyfuncs.vsl, 4, 73, 16797824
Aarch64, aamanyfuncs.vsl, 5, 74, 17198288
Aarch64, aamanyfuncs.vsl, 6, 74, 16987648
Aarch64, aamanyfuncs.vsl, 7, 81, 17224323
Aarch64, aamanyfuncs.vsl, 8, 74, 17142104
Aarch64, aamanyfuncs.vsl, 9, 75, 17340244
Aarch64, aamanyfuncs.vsl, 10, 84, 17216337
Aarch64, aamanyfuncs.vsl, 11, 80, 16309577
Aarch64, aamanyfuncs.vsl, 12, 80, 16675804
Aarch64, aamanyfuncs.vsl, 13, 81, 15925547
Aarch64, aamanyfuncs.vsl, 14, 80, 15821680
Aarch64, aamanyfuncs.vsl, 15, 75, 15448533
Aarch64, aamanyfuncs.vsl, 16, 79, 15447889
ASTOptimisation, aamanyfuncs.vsl, 1, 134, 8926580
ASTOptimisation, aamanyfuncs.vsl, 2, 138, 8530756
ASTOptimisation, aamanyfuncs.vsl, 3, 139, 8509808
ASTOptimisation, aamanyfuncs.vsl, 4, 138, 8862971
ASTOptimisation, aamanyfuncs.vsl, 5, 136, 8605013
ASTOptimisation, aamanyfuncs.vsl, 6, 136, 8795389
ASTOptimisation, aamanyfuncs.vsl, 7, 134, 8862248
ASTOptimisation, aamanyfuncs.vsl, 8, 135, 8916899
ASTOptimisation, aamanyfuncs.vsl, 9, 133, 8990146
ASTOptimisation, aamanyfuncs.vsl, 10, 133, 9113969
ASTOptimisation, aamanyfuncs.vsl, 11, 130, 9089605
ASTOptimisation, aamanyfuncs.vsl, 12, 130, 9275212
ASTOptimisation, aamanyfuncs.vsl, 13, 132, 9097721
ASTOptimisation, aamanyfuncs.vsl, 14, 128, 9239324
ASTOptimisation, aamanyfuncs.vsl, 15, 127, 9422543
ASTOptimisation, aamanyfuncs.vsl, 16, 127, 9379509
LIRGeneration, aamanyfuncs.vsl, 1, 834, 1576556
LIRGeneration, aamanyfuncs.vsl, 2, 946, 1198232
LIRGeneration, aamanyfuncs.vsl, 3, 1040, 1164831
LIRGeneration, aamanyfuncs.vsl, 4, 1036, 1200676
LIRGeneration, aamanyfuncs.vsl, 5, 1023, 1167084
LIRGeneration, aamanyfuncs.vsl, 6, 1033, 1138968
LIRGeneration, aamanyfuncs.vsl, 7, 1034, 1190615
LIRGeneration, aamanyfuncs.vsl, 8, 937, 1194034
LIRGeneration, aamanyfuncs.vsl, 9, 981, 1208147
LIRGeneration, aamanyfuncs.vsl, 10, 1039, 1221395
LIRGeneration, aamanyfuncs.vsl, 11, 859, 1229361
LIRGeneration, aamanyfuncs.vsl, 12, 973, 1252032
LIRGeneration, aamanyfuncs.vsl, 13, 920, 1234986
LIRGeneration, aamanyfuncs.vsl, 14, 916, 1254004
LIRGeneration, aamanyfuncs.vsl, 15, 984, 1238378
LIRGeneration, aamanyfuncs.vsl, 16, 894, 1246766
RegisterAllocation, aamanyfuncs.vsl, 1, 176, 6656478
RegisterAllocation, aamanyfuncs.vsl, 2, 300, 4091932
RegisterAllocation, aamanyfuncs.vsl, 3, 367, 3277914
RegisterAllocation, aamanyfuncs.vsl, 4, 366, 3093633
RegisterAllocation, aamanyfuncs.vsl, 5, 397, 3036365
RegisterAllocation, aamanyfuncs.vsl, 6, 405, 2885572
RegisterAllocation, aamanyfuncs.vsl, 7, 394, 2993545
RegisterAllocation, aamanyfuncs.vsl, 8, 416, 2764071
```

```
RegisterAllocation, aamanyfuncs.vsl, 9, 490, 2514620
RegisterAllocation, aamanyfuncs.vsl, 10, 531, 2203247
RegisterAllocation, aamanyfuncs.vsl, 11, 546, 2201906
RegisterAllocation, aamanyfuncs.vsl, 12, 513, 2340934
RegisterAllocation, aamanyfuncs.vsl, 13, 529, 2265518
RegisterAllocation, aamanyfuncs.vsl, 14, 570, 2111049
RegisterAllocation, aamanyfuncs.vsl, 15, 550, 2166476
RegisterAllocation, aamanyfuncs.vsl, 16, 556, 2156612
AssemblerGeneration, aamanyfuncs.vsl, 1, 286, 4792530
AssemblerGeneration, aamanyfuncs.vsl, 2, 412, 3740891
AssemblerGeneration, aamanyfuncs.vsl, 3, 409, 3734341
AssemblerGeneration, aamanyfuncs.vsl, 4, 480, 3538470
AssemblerGeneration, aamanyfuncs.vsl, 5, 475, 3543883
AssemblerGeneration, aamanyfuncs.vsl, 6, 457, 3686045
AssemblerGeneration, aamanyfuncs.vsl, 7, 434, 3597709
AssemblerGeneration, aamanyfuncs.vsl, 8, 434, 3803372
AssemblerGeneration, aamanyfuncs.vsl, 9, 454, 3624859
AssemblerGeneration, aamanyfuncs.vsl, 10, 458, 3634458
AssemblerGeneration, aamanyfuncs.vsl, 11, 445, 3741628
AssemblerGeneration, aamanyfuncs.vsl, 12, 438, 3695539
AssemblerGeneration, aamanyfuncs.vsl, 13, 450, 3708413
AssemblerGeneration, aamanyfuncs.vsl, 14, 440, 3714696
AssemblerGeneration, aamanyfuncs.vsl, 15, 446, 3712444
AssemblerGeneration, aamanyfuncs.vsl, 16, 435, 3675316
Aarch64, harder.vsl, 1, 1868, 633355
Aarch64, harder.vsl, 2, 1309, 879619
Aarch64, harder.vsl, 3, 1416, 858854
Aarch64, harder.vsl, 4, 1311, 825417
Aarch64, harder.vsl, 5, 1424, 946427
Aarch64, harder.vsl, 6, 1414, 832041
Aarch64, harder.vsl, 7, 1334, 851029
Aarch64, harder.vsl, 8, 1203, 868102
Aarch64, harder.vsl, 9, 1392, 840072
Aarch64, harder.vsl, 10, 1452, 871571
Aarch64, harder.vsl, 11, 1382, 873129
Aarch64, harder.vsl, 12, 1416, 986147
Aarch64, harder.vsl, 13, 1326, 949924
Aarch64, harder.vsl, 14, 1340, 868671
Aarch64, harder.vsl, 15, 1401, 872474
Aarch64, harder.vsl, 16, 1419, 869271
ASTOptimisation, harder.vsl, 1, 5144, 225990
ASTOptimisation, harder.vsl, 2, 4974, 246654
ASTOptimisation, harder.vsl, 3, 5229, 270612
ASTOptimisation, harder.vsl, 4, 5119, 260204
ASTOptimisation, harder.vsl, 5, 4977, 245910
ASTOptimisation, harder.vsl, 6, 5718, 234808
ASTOptimisation, harder.vsl, 7, 4437, 249376
ASTOptimisation, harder.vsl, 8, 5070, 249422
ASTOptimisation, harder.vsl, 9, 4700, 245611
ASTOptimisation, harder.vsl, 10, 4462, 249548
ASTOptimisation, harder.vsl, 11, 4927, 249055
ASTOptimisation, harder.vsl, 12, 4999, 257634
ASTOptimisation, harder.vsl, 13, 4422, 253210
ASTOptimisation, harder.vsl, 14, 4962, 247632
ASTOptimisation, harder.vsl, 15, 4502, 251497
ASTOptimisation, harder.vsl, 16, 4704, 245413
LIRGeneration, harder.vsl, 1, 20618, 51768
LIRGeneration, harder.vsl, 2, 7186, 160614
LIRGeneration, harder.vsl, 3, 7330, 164014
```

```
LIRGeneration, harder.vsl, 4, 6680, 163428
LIRGeneration, harder.vsl, 5, 6558, 172527
LIRGeneration, harder.vsl, 6, 6866, 161904
LIRGeneration, harder.vsl, 7, 7137, 192332
LIRGeneration, harder.vsl, 8, 6925, 168725
LIRGeneration, harder.vsl, 9, 7364, 160959
LIRGeneration, harder.vsl, 10, 6646, 164911
LIRGeneration, harder.vsl, 11, 6936, 166258
LIRGeneration, harder.vsl, 12, 6736, 163319
LIRGeneration, harder.vsl, 13, 7358, 169337
LIRGeneration, harder.vsl, 14, 6358, 166832
LIRGeneration, harder.vsl, 15, 6919, 166033
LIRGeneration, harder.vsl, 16, 6774, 162333
RegisterAllocation, harder.vsl, 1, 15109, 75753
RegisterAllocation, harder.vsl, 2, 7081, 169324
RegisterAllocation, harder.vsl, 3, 8629, 148062
RegisterAllocation, harder.vsl, 4, 7131, 156394
RegisterAllocation, harder.vsl, 5, 7411, 154205
RegisterAllocation, harder.vsl, 6, 8186, 154534
RegisterAllocation, harder.vsl, 7, 7227, 162626
RegisterAllocation, harder.vsl, 8, 7249, 164542
RegisterAllocation, harder.vsl, 9, 7914, 148540
RegisterAllocation, harder.vsl, 10, 8340, 150903
RegisterAllocation, harder.vsl, 11, 8031, 146644
RegisterAllocation, harder.vsl, 12, 9090, 154118
RegisterAllocation, harder.vsl, 13, 10000, 151504
RegisterAllocation, harder.vsl, 14, 8325, 161830
RegisterAllocation, harder.vsl, 15, 10000, 156023
RegisterAllocation, harder.vsl, 16, 9867, 157191
AssemblerGeneration, harder.vsl, 1, 4147, 5082879
AssemblerGeneration, harder.vsl, 2, 3336, 5587537
AssemblerGeneration, harder.vsl, 3, 3266, 5532561
AssemblerGeneration, harder.vsl, 4, 3991, 6770002
AssemblerGeneration, harder.vsl, 5, 3416, 5760571
AssemblerGeneration, harder.vsl, 6, 3804, 6307272
AssemblerGeneration, harder.vsl, 7, 3397, 5745819
AssemblerGeneration, harder.vsl, 8, 3900, 6488831
AssemblerGeneration, harder.vsl, 9, 3901, 6606566
AssemblerGeneration, harder.vsl, 10, 3520, 5889990
AssemblerGeneration, harder.vsl, 11, 2637, 4622078
AssemblerGeneration, harder.vsl, 12, 3894, 6517261
AssemblerGeneration, harder.vsl, 13, 3603, 6111222
AssemblerGeneration, harder.vsl, 14, 3429, 5803609
AssemblerGeneration, harder.vsl, 15, 3294, 5674743
AssemblerGeneration, harder.vsl, 16, 3860, 6357579
Aarch64, if2.vsl, 1, 3344, 360097
Aarch64, if2.vsl, 2, 2191, 528281
Aarch64, if2.vsl, 3, 2274, 532635
Aarch64, if2.vsl, 4, 2208, 519216
Aarch64, if2.vsl, 5, 2348, 577018
Aarch64, if2.vsl, 6, 2227, 538787
Aarch64, if2.vsl, 7, 2155, 535832
Aarch64, if2.vsl, 8, 2234, 575288
Aarch64, if2.vsl, 9, 2056, 550866
Aarch64, if2.vsl, 10, 2226, 538330
Aarch64, if2.vsl, 11, 2128, 534875
Aarch64, if2.vsl, 12, 2173, 555026
Aarch64, if2.vsl, 13, 2088, 589795
Aarch64, if2.vsl, 14, 2245, 548679
```

```
Aarch64, if2.vsl, 15, 2052, 557815
Aarch64, if2.vsl, 16, 2130, 568685
ASTOptimisation, if2.vsl, 1, 15174, 73038
ASTOptimisation, if2.vsl, 2, 10000, 111928
ASTOptimisation, if2.vsl, 3, 10000, 102128
ASTOptimisation, if2.vsl, 4, 12252, 87664
ASTOptimisation, if2.vsl, 5, 12282, 92399
ASTOptimisation, if2.vsl, 6, 12403, 95024
ASTOptimisation, if2.vsl, 7, 12288, 86443
ASTOptimisation, if2.vsl, 8, 9860, 103654
ASTOptimisation, if2.vsl, 9, 10000, 102910
ASTOptimisation, if2.vsl, 10, 12356, 87904
ASTOptimisation, if2.vsl, 11, 10000, 100359
ASTOptimisation, if2.vsl, 12, 9747, 103093
ASTOptimisation, if2.vsl, 13, 12369, 83880
ASTOptimisation, if2.vsl, 14, 13074, 87649
ASTOptimisation, if2.vsl, 15, 12201, 86976
ASTOptimisation, if2.vsl, 16, 12186, 85710
LIRGeneration, if2.vsl, 1, 42235, 26655
LIRGeneration, if2.vsl, 2, 12901, 90758
LIRGeneration, if2.vsl, 3, 13077, 91509
LIRGeneration, if2.vsl, 4, 13273, 77761
LIRGeneration, if2.vsl, 5, 12888, 89820
LIRGeneration, if2.vsl, 6, 12795, 89460
LIRGeneration, if2.vsl, 7, 12778, 89148
LIRGeneration, if2.vsl, 8, 12972, 95971
LIRGeneration, if2.vsl, 9, 12970, 92732
LIRGeneration, if2.vsl, 10, 12442, 88889
LIRGeneration, if2.vsl, 11, 12909, 90010
LIRGeneration, if2.vsl, 12, 12477, 88491
LIRGeneration, if2.vsl, 13, 12997, 91127
LIRGeneration, if2.vsl, 14, 12973, 90354
LIRGeneration, if2.vsl, 15, 12702, 90039
LIRGeneration, if2.vsl, 16, 13024, 91116
RegisterAllocation, if2.vsl, 1, 30324, 39458
RegisterAllocation, if2.vsl, 2, 10000, 114147
RegisterAllocation, if2.vsl, 3, 10000, 111369
RegisterAllocation, if2.vsl, 4, 10000, 117331
RegisterAllocation, if2.vsl, 5, 10000, 115199
RegisterAllocation, if2.vsl, 6, 10000, 121005
RegisterAllocation, if2.vsl, 7, 10000, 113113
RegisterAllocation, if2.vsl, 8, 10000, 106727
RegisterAllocation, if2.vsl, 9, 9162, 114506
RegisterAllocation, if2.vsl, 10, 10000, 107074
RegisterAllocation, if2.vsl, 11, 10000, 110236
RegisterAllocation, if2.vsl, 12, 10000, 113363
RegisterAllocation, if2.vsl, 13, 10000, 115211
RegisterAllocation, if2.vsl, 14, 10000, 109722
RegisterAllocation, if2.vsl, 15, 10000, 115951
RegisterAllocation, if2.vsl, 16, 10000, 113475
AssemblerGeneration, if2.vsl, 1, 2673, 7317540
AssemblerGeneration, if2.vsl, 2, 3030, 9662654
AssemblerGeneration, if2.vsl, 3, 3165, 9814580
AssemblerGeneration, if2.vsl, 4, 1996, 6995814
AssemblerGeneration, if2.vsl, 5, 2749, 8801730
AssemblerGeneration, if2.vsl, 6, 2948, 9767194
AssemblerGeneration, if2.vsl, 7, 2730, 8759913
AssemblerGeneration, if2.vsl, 8, 3001, 9532706
AssemblerGeneration, if2.vsl, 9, 3092, 9675535
```

```
AssemblerGeneration, if2.vsl, 10, 3031, 9546670
AssemblerGeneration, if2.vsl, 11, 2961, 9095515
AssemblerGeneration, if2.vsl, 12, 3027, 9604543
AssemblerGeneration, if2.vsl, 13, 3076, 9565191
AssemblerGeneration, if2.vsl, 14, 2948, 9330891
AssemblerGeneration, if2.vsl, 15, 3016, 9847074
AssemblerGeneration, if2.vsl, 16, 2983, 9606044
Aarch64, multi_hello.vsl, 1, 1586, 1122376
Aarch64, multi_hello.vsl, 2, 982, 1262472
Aarch64, multi_hello.vsl, 3, 831, 1391993
Aarch64, multi_hello.vsl, 4, 870, 1373684
Aarch64, multi_hello.vsl, 5, 866, 1397914
Aarch64, multi_hello.vsl, 6, 846, 1358006
Aarch64, multi_hello.vsl, 7, 837, 1424494
Aarch64, multi_hello.vsl, 8, 812, 1330660
Aarch64, multi_hello.vsl, 9, 834, 1397580
Aarch64, multi_hello.vsl, 10, 852, 1339433
Aarch64, multi_hello.vsl, 11, 842, 1363472
Aarch64, multi_hello.vsl, 12, 868, 1313532
Aarch64, multi_hello.vsl, 13, 890, 1379670
Aarch64, multi_hello.vsl, 14, 852, 1335312
Aarch64, multi_hello.vsl, 15, 900, 1511420
Aarch64, multi_hello.vsl, 16, 832, 1355676
ASTOptimisation, multi_hello.vsl, 1, 3390, 348223
ASTOptimisation, multi_hello.vsl, 2, 2804, 416507
ASTOptimisation, multi_hello.vsl, 3, 2635, 426759
ASTOptimisation, multi_hello.vsl, 4, 2600, 441850
ASTOptimisation, multi_hello.vsl, 5, 2760, 433729
ASTOptimisation, multi_hello.vsl, 6, 2755, 423430
ASTOptimisation, multi_hello.vsl, 7, 2682, 446192
ASTOptimisation, multi_hello.vsl, 8, 2692, 442396
ASTOptimisation, multi_hello.vsl, 9, 2667, 427293
ASTOptimisation, multi_hello.vsl, 10, 2634, 421043
ASTOptimisation, multi_hello.vsl, 11, 2570, 426817
ASTOptimisation, multi_hello.vsl, 12, 2724, 425744
ASTOptimisation, multi_hello.vsl, 13, 2637, 424658
ASTOptimisation, multi_hello.vsl, 14, 2728, 443271
ASTOptimisation, multi_hello.vsl, 15, 2858, 435308
ASTOptimisation, multi_hello.vsl, 16, 2624, 440505
LIRGeneration, multi_hello.vsl, 1, 10000, 100666
LIRGeneration, multi_hello.vsl, 2, 6412, 178506
LIRGeneration, multi_hello.vsl, 3, 5214, 219347
LIRGeneration, multi_hello.vsl, 4, 5198, 225935
LIRGeneration, multi_hello.vsl, 5, 5192, 216947
LIRGeneration, multi_hello.vsl, 6, 5265, 226889
LIRGeneration, multi_hello.vsl, 7, 4972, 211605
LIRGeneration, multi_hello.vsl, 8, 4842, 230333
LIRGeneration, multi_hello.vsl, 9, 4840, 226227
LIRGeneration, multi_hello.vsl, 10, 5341, 236252
LIRGeneration, multi_hello.vsl, 11, 5538, 216233
LIRGeneration, multi_hello.vsl, 12, 5272, 220995
LIRGeneration, multi_hello.vsl, 13, 4855, 216188
LIRGeneration, multi_hello.vsl, 14, 5005, 224867
LIRGeneration, multi_hello.vsl, 15, 5197, 233683
LIRGeneration, multi_hello.vsl, 16, 5510, 219422
RegisterAllocation, multi_hello.vsl, 1, 5755, 213065
RegisterAllocation, multi_hello.vsl, 2, 4243, 260062
RegisterAllocation, multi_hello.vsl, 3, 3618, 297136
RegisterAllocation, multi_hello.vsl, 4, 3919, 299026
```

```
RegisterAllocation, multi_hello.vsl, 5, 4094, 294226
RegisterAllocation, multi_hello.vsl, 6, 3824, 288816
RegisterAllocation, multi_hello.vsl, 7, 4074, 292650
RegisterAllocation, multi_hello.vsl, 8, 4090, 294734
RegisterAllocation, multi_hello.vsl, 9, 3927, 319155
RegisterAllocation, multi_hello.vsl, 10, 4128, 310777
RegisterAllocation, multi_hello.vsl, 11, 3930, 317086
RegisterAllocation, multi_hello.vsl, 12, 3982, 290385
RegisterAllocation, multi_hello.vsl, 13, 4386, 293293
RegisterAllocation, multi_hello.vsl, 14, 5048, 322916
RegisterAllocation, multi_hello.vsl, 15, 3980, 308419
RegisterAllocation, multi_hello.vsl, 16, 4183, 297755
AssemblerGeneration, multi_hello.vsl, 1, 1858, 2656417
AssemblerGeneration, multi_hello.vsl, 2, 2300, 4229907
AssemblerGeneration, multi_hello.vsl, 3, 2282, 4369128
AssemblerGeneration, multi_hello.vsl, 4, 2296, 4247503
AssemblerGeneration, multi_hello.vsl, 5, 2374, 4350418
AssemblerGeneration, multi_hello.vsl, 6, 2416, 4537417
AssemblerGeneration, multi_hello.vsl, 7, 2340, 4308509
AssemblerGeneration, multi_hello.vsl, 8, 2250, 4185494
AssemblerGeneration, multi_hello.vsl, 9, 2397, 4363041
AssemblerGeneration, multi_hello.vsl, 10, 2296, 4211804
AssemblerGeneration, multi_hello.vsl, 11, 2166, 4036281
AssemblerGeneration, multi_hello.vsl, 12, 2234, 4138428
AssemblerGeneration, multi_hello.vsl, 13, 2713, 4891297
AssemblerGeneration, multi_hello.vsl, 14, 2727, 4946471
AssemblerGeneration, multi_hello.vsl, 15, 2236, 4113731
AssemblerGeneration, multi_hello.vsl, 16, 2229, 4122315
```

Appendix D: Benchmark results hashmap raw

```
Callsign, Operation, n, b.N, Results (ns/op)
crc32p, Insert, 10, 339699, 3755
crc32p, Lookup, 10, 2372703, 475
crc32p, Remove, 10, 252591, 5153
crc32re, Insert, 10, 784000, 1605
crc32re, Lookup, 10, 2409684, 472
crc32re, Remove, 10, 405198, 3532
crc32opt, Insert, 10, 1177317, 976
crc32opt, Lookup, 10, 2560135, 463
crc32opt, Remove, 10, 312448, 4679
djb2opt, Insert, 10, 1529737, 790
djb2opt, Lookup, 10, 2563114, 469
djb2opt, Remove, 10, 290876, 4262
goh, Insert, 10, 1346206, 872
goh, Lookup, 10, 5438748, 204
goh, Remove, 10, 462085, 2431
```

Appendix E: Profiling data CPU

CPU profiling data, from *aamanyfuncs.vsl* source file, generated by *BenchmarkLIRGeneration* and *BenchmarkAssemblerGeneration* on the vm callsign system, with one, two, four and 16 worker goroutines using the vslc *-t* flag. Only top 30 results are listed for brevity.

LIR generation -t = 1

```
Showing nodes accounting for 1.54s, 75.12% of 2.05s total
Dropped 74 nodes (cum <= 0.01s)
Showing top 30 nodes out of 123
      flat  flat%   sum%        cum   cum%
      0.14s  6.83%  6.83%       0.47s 22.93%  runtime.mallocgc
      0.11s  5.37% 12.20%       0.13s  6.34%  runtime.mapaccess2_faststr
      0.10s  4.88% 17.07%       0.29s 14.15%  runtime.scanobject
      0.09s  4.39% 21.46%       0.52s 25.37%  runtime.gcDrain
      0.08s  3.90% 25.37%       0.09s  4.39%  runtime.heapBitsSetType
      0.07s  3.41% 28.78%       0.07s  3.41%  runtime.futex
      0.07s  3.41% 32.20%       0.07s  3.41%  runtime.memclrNoHeapPointers
      0.07s  3.41% 35.61%       0.07s  3.41%  sync.(*Mutex).Unlock
      0.07s  3.41% 39.02%       1.14s 55.61%  vslc/src/ir/lir.genAssign
      0.07s  3.41% 42.44%       0.13s  6.34%  vslc/src/util.(*Stack).Size
      0.06s  2.93% 45.37%       0.08s  3.90%  runtime.getempty
      0.06s  2.93% 48.29%       0.06s  2.93%  runtime.markBits.isMarked (inline)
      0.06s  2.93% 51.22%       0.07s  3.41%  vslc/src/util.(*Stack).Get
      0.04s  1.95% 53.17%       0.04s  1.95%  runtime.(*lfstack).pop (inline)
      0.04s  1.95% 55.12%       0.04s  1.95%  runtime.findObject
      0.04s  1.95% 57.07%       0.13s  6.34%  runtime.greyobject
      0.04s  1.95% 59.02%       0.23s 11.22%
vslc/src/ir/lir.(*Block).CreateStore
      0.03s  1.46% 60.49%       0.03s  1.46%  runtime.heapBits.bits (inline)
      0.03s  1.46% 61.95%       0.05s  2.44%  runtime.lock2
      0.03s  1.46% 63.41%       0.03s  1.46%  runtime.madvise
      0.03s  1.46% 64.88%       0.03s  1.46%  runtime.nextFreeFast (inline)
      0.03s  1.46% 66.34%       0.03s  1.46%  runtime.releasem (inline)
      0.03s  1.46% 67.80%       0.17s  8.29%  vslc/src/ir/lir.(*Block).CreateLoad
      0.03s  1.46% 69.27%       0.43s 20.98%  vslc/src/ir/lir.genExpression
      0.02s  0.98% 70.24%       0.02s  0.98%  aeshashbody
      0.02s  0.98% 71.22%       0.03s  1.46%  fmt.(*pp).printArg
      0.02s  0.98% 72.20%       0.02s  0.98%  runtime.(*gcBits).bitp (inline)
      0.02s  0.98% 73.17%       0.02s  0.98%  runtime.nanotime
      0.02s  0.98% 74.15%       0.02s  0.98%  runtime.pageIndexOf (inline)
      0.02s  0.98% 75.12%       0.02s  0.98%  runtime.publicationBarrier
```

LIR generation -t = 2

```
Showing nodes accounting for 2.57s, 74.93% of 3.43s total
Dropped 79 nodes (cum <= 0.02s)
Showing top 30 nodes out of 144
      flat  flat%   sum%        cum   cum%
      0.52s 15.16% 15.16%       1.31s 38.19%  unicode.init
      0.27s  7.87% 23.03%       0.44s 12.83%  runtime.main
      0.18s  5.25% 28.28%       0.20s  5.83%
reflect.(*interfaceType).MethodByName
      0.15s  4.37% 32.65%       0.20s  5.83%  runtime.recordspan
      0.11s  3.21% 35.86%       0.16s  4.66%  runtime.(*pageAlloc).find
      0.10s  2.92% 38.78%       0.10s  2.92%  runtime.acquireSudog
      0.09s  2.62% 41.40%       0.09s  2.62%  runtime.(*pageCache).flush
      0.08s  2.33% 43.73%       0.17s  4.96%  runtime.handoff
      0.08s  2.33% 46.06%       0.08s  2.33%  runtime.trygetfull
      0.07s  2.04% 48.10%       0.40s 11.66%
vslc/src/ir/lir.(*Block).CreateStore
      0.07s  2.04% 50.15%       0.97s 28.28%  vslc/src/ir/lir.genExpression
      0.06s  1.75% 51.90%       0.16s  4.66%  runtime.getempty
      0.06s  1.75% 53.64%       0.40s 11.66%  vslc/src/ir/lir.(*Block).CreateLoad
      0.06s  1.75% 55.39%       1.99s 58.02%  vslc/src/ir/lir.genAssign
      0.05s  1.46% 56.85%       0.05s  1.46%
_cgo_5c1398b82605_Cfunc_LLVMBuildPhi
      0.05s  1.46% 58.31%       0.05s  1.46%  runtime.gcSetTriggerRatio
      0.05s  1.46% 59.77%       0.13s  3.79%  runtime.hexdumpWords
      0.05s  1.46% 61.22%       0.05s  1.46%  runtime.mProf_Malloc
      0.05s  1.46% 62.68%       0.10s  2.92%  runtime.printuint
      0.05s  1.46% 64.14%       2.22s 64.72%  vslc/src/ir/lir.gen
      0.04s  1.17% 65.31%       0.04s  1.17%
_cgo_5c1398b82605_Cfunc_LLVMBuildRet
      0.04s  1.17% 66.47%       0.04s  1.17%  internal/reflectlite.(*rtype).Elem
      0.04s  1.17% 67.64%       0.79s 23.03%  runtime.(*mheap).init
      0.04s  1.17% 68.80%       0.04s  1.17%  runtime.mProf_Flush
      0.04s  1.17% 69.97%       0.04s  1.17%  runtime.stkbucket
      0.04s  1.17% 71.14%       0.20s  5.83%  runtime.stopTheWorldWithSema
      0.04s  1.17% 72.30%       0.04s  1.17%  sync.(*poolChain).pushHead
      0.03s  0.87% 73.18%       0.03s  0.87%
runtime.(*gcControllerState).findRunnableGCWorker
      0.03s  0.87% 74.05%       0.04s  1.17%  runtime.(*pageAlloc).sysGrow
      0.03s  0.87% 74.93%       0.03s  0.87%  runtime.(*pallocData).allocRange
```

LIR generation -t = 4

```
Showing nodes accounting for 2.62s, 64.53% of 4.06s total
Dropped 155 nodes (cum <= 0.02s)
Showing top 30 nodes out of 148
      flat  flat%   sum%        cum   cum%
      0.27s  6.65%  6.65%       1.18s 29.06%  runtime.mallocgc
      0.25s  6.16% 12.81%       0.25s  6.16%  runtime.futex
      0.20s  4.93% 17.73%       0.32s  7.88%  runtime.heapBitsSetType
      0.20s  4.93% 22.66%       0.53s 13.05%  runtime.scanobject
      0.16s  3.94% 26.60%       0.16s  3.94%  runtime.memclrNoHeapPointers
      0.11s  2.71% 29.31%       0.17s  4.19%  runtime.findObject
      0.11s  2.71% 32.02%       0.11s  2.71%  runtime.madvise
      0.11s  2.71% 34.73%       2.19s 53.94%  vslc/src/ir/lir.genAssign
      0.09s  2.22% 36.95%       0.09s  2.22%  runtime.markBits.isMarked (inline)
      0.08s  1.97% 38.92%       0.18s  4.43%  runtime.mapaccess2_faststr
      0.08s  1.97% 40.89%       0.10s  2.46%  sync.(*Mutex).Unlock
      0.08s  1.97% 42.86%       0.58s 14.29%  vslc/src/ir/lir.(*Block).CreateLoad
      0.07s  1.72% 44.58%       0.73s 17.98%  runtime.gcDrain
      0.07s  1.72% 46.31%       0.14s  3.45%  runtime.greyobject
      0.07s  1.72% 48.03%       0.08s  1.97%  runtime.heapBitsForAddr
(partial-inline)
      0.07s  1.72% 49.75%       0.94s 23.15%  vslc/src/ir/lir.genExpression
      0.06s  1.48% 51.23%       0.06s  1.48%  runtime.nextFreeFast (inline)
      0.06s  1.48% 52.71%       0.12s  2.96%  vslc/src/util.(*Stack).Size
      0.05s  1.23% 53.94%       0.05s  1.23%  memeqbody
      0.05s  1.23% 55.17%       0.05s  1.23%  runtime.osyield
      0.05s  1.23% 56.40%       0.37s  9.11%
vslc/src/ir/lir.(*Block).CreateStore
      0.04s  0.99% 57.39%       0.04s  0.99%  aeshashbody
      0.04s  0.99% 58.37%       0.04s  0.99%  runtime.(*lfstack).push
      0.04s  0.99% 59.36%       0.05s  1.23%  runtime.getempty
      0.04s  0.99% 60.34%       0.04s  0.99%  runtime.memmove
      0.04s  0.99% 61.33%       0.26s  6.40%
vslc/src/ir/lir.(*Block).CreateConstantInt
      0.04s  0.99% 62.32%       0.18s  4.43%
vslc/src/ir/lir.(*Block).CreateFunctionCall
      0.03s  0.74% 63.05%       0.10s  2.46%  fmt.(*pp).doPrintf
      0.03s  0.74% 63.79%       0.07s  1.72%  fmt.(*pp).printArg
      0.03s  0.74% 64.53%       0.03s  0.74%  runtime.(*gcBitsArena).tryAlloc
(inline)
```

LIR generation -t = 16

```
Showing nodes accounting for 3.15s, 60.93% of 5.17s total
Dropped 145 nodes (cum <= 0.03s)
Showing top 30 nodes out of 169
      flat  flat%   sum%        cum   cum%
      0.28s  5.42%  5.42%       0.42s  8.12%  runtime.heapBitsSetType
      0.26s  5.03% 10.44%       1.38s 26.69%  runtime.mallocgc
      0.26s  5.03% 15.47%       0.79s 15.28%  runtime.scanobject
      0.25s  4.84% 20.31%       0.25s  4.84%  runtime.futex
      0.20s  3.87% 24.18%       0.20s  3.87%  runtime.madvise
      0.16s  3.09% 27.27%       0.16s  3.09%  runtime.memclrNoHeapPointers
      0.14s  2.71% 29.98%       0.20s  3.87%  runtime.findObject
      0.11s  2.13% 32.11%       0.32s  6.19%  runtime.greyobject
      0.10s  1.93% 34.04%       1.02s 19.73%  runtime.gcDrain
      0.09s  1.74% 35.78%       0.14s  2.71%  runtime.mapaccess2_faststr
      0.09s  1.74% 37.52%       0.09s  1.74%  runtime.pageIndexOf (inline)
      0.09s  1.74% 39.26%       0.23s  4.45%  runtime.pcvalue
      0.09s  1.74% 41.01%       1.02s 19.73%  vslc/src/ir/lir.genExpression
      0.08s  1.55% 42.55%       0.08s  1.55%  runtime.markBits.isMarked (inline)
      0.08s  1.55% 44.10%       0.12s  2.32%  runtime.step
      0.08s  1.55% 45.65%       2.78s 53.77%  vslc/src/ir/lir.gen
      0.07s  1.35% 47.00%       0.48s  9.28%
vslc/src/ir/lir.(*Block).CreateConstantInt
      0.07s  1.35% 48.36%       0.16s  3.09%
vslc/src/ir/lir.(*Block).createArithmeticInstruction
      0.07s  1.35% 49.71%       0.09s  1.74%  vslc/src/util.(*Stack).Size
      0.06s  1.16% 50.87%       0.06s  1.16%  runtime.(*lfstack).push
      0.06s  1.16% 52.03%       0.06s  1.16%  runtime.heapBitsForAddr (inline)
      0.06s  1.16% 53.19%       0.06s  1.16%  runtime.procyield
      0.05s  0.97% 54.16%       0.12s  2.32%  fmt.(*pp).doPrintf
      0.05s  0.97% 55.13%       0.05s  0.97%  runtime.getpid
      0.05s  0.97% 56.09%       0.05s  0.97%  runtime.heapBits.bits (inline)
      0.05s  0.97% 57.06%       0.05s  0.97%  runtime.nextFreeFast (inline)
      0.05s  0.97% 58.03%       0.05s  0.97%  runtime.tgkill
      0.05s  0.97% 58.99%       0.10s  1.93%  runtime.wbBufFlush1
      0.05s  0.97% 59.96%       0.06s  1.16%  sync.(*Mutex).Unlock
(partial-inline)
      0.05s  0.97% 60.93%       0.45s  8.70%  vslc/src/ir/lir.(*Block).CreateLoad
```

Assembler generation -t = 1

```
Showing nodes accounting for 1970ms, 74.62% of 2640ms total
Dropped 78 nodes (cum <= 13.20ms)
Showing top 30 nodes out of 114
      flat  flat%   sum%        cum   cum%
      190ms  7.20%  7.20%       940ms 35.61%  fmt.(*pp).doPrintf
      180ms  6.82% 14.02%       540ms 20.45%  runtime.scanobject
      160ms  6.06% 20.08%       160ms  6.06%  runtime.memmove
      140ms  5.30% 25.38%       330ms 12.50%  runtime.mallocgc
      100ms  3.79% 29.17%       620ms 23.48%  fmt.(*pp).printArg
      80ms  3.03% 32.20%        150ms  5.68%  fmt.(*buffer).writeString (inline)
      80ms  3.03% 35.23%        80ms  3.03%  runtime.markBits.isMarked (inline)
      80ms  3.03% 38.26%        150ms  5.68%  strconv.appendQuotedWith
      70ms  2.65% 40.91%        130ms  4.92%  runtime.findObject
      60ms  2.27% 43.18%        220ms  8.33%  fmt.(*pp).handleMethods
      60ms  2.27% 45.45%  1490ms 56.44%  vslc/src/backend/arm.genFunction
      50ms  1.89% 47.35%        60ms  2.27%  fmt.(*fmt).fmtInteger
      50ms  1.89% 49.24%        90ms  3.41%  fmt.newPrinter
      50ms  1.89% 51.14%        50ms  1.89%  runtime.(*itabTableType).find
      50ms  1.89% 53.03%        700ms 26.52%  runtime.gcDrain
      50ms  1.89% 54.92%        50ms  1.89%  runtime.heapBits.next (inline)
      50ms  1.89% 56.82%        50ms  1.89%  runtime.pageIndexOf (inline)
      50ms  1.89% 58.71%        70ms  2.65%  strconv.appendEscapedRune
      50ms  1.89% 60.61%        240ms  9.09%
vslc/src/backend/arm.genFunctionCall
      40ms  1.52% 62.12%        80ms  3.03%  fmt.(*fmt).fmtS
      40ms  1.52% 63.64%        40ms  1.52%  runtime.nextFreeFast (inline)
      40ms  1.52% 65.15%        60ms  2.27%  runtime.scanblock
      40ms  1.52% 66.67%        40ms  1.52%  sync.(*Pool).pin
      30ms  1.14% 67.80%        30ms  1.14%  runtime.(*gcBits).bitp (inline)
      30ms  1.14% 68.94%        30ms  1.14%  runtime.(*mspan).divideByElemSize
(inline)
      30ms  1.14% 70.08%        30ms  1.14%  runtime.futex
      30ms  1.14% 71.21%        80ms  3.03%  runtime.getitab
      30ms  1.14% 72.35%        180ms  6.82%  runtime.greyobject
      30ms  1.14% 73.48%        30ms  1.14%  runtime.heapBits.bits (inline)
      30ms  1.14% 74.62%        50ms  1.89%  sync.(*Pool).Put
```

Assembler generation -t = 2

```
Showing nodes accounting for 3.34s, 73.25% of 4.56s total
Dropped 116 nodes (cum <= 0.02s)
Showing top 30 nodes out of 132
      flat  flat%   sum%        cum   cum%
      0.31s  6.80%  6.80%       1.55s 33.99%  fmt.(*pp).doPrintf
      0.28s  6.14% 12.94%       0.70s 15.35%  runtime.mallocgc
      0.28s  6.14% 19.08%       0.95s 20.83%  runtime.scanobject
      0.26s  5.70% 24.78%       0.26s  5.70%  runtime.memmove
      0.22s  4.82% 29.61%       0.22s  4.82%  runtime.markBits.isMarked (inline)
      0.18s  3.95% 33.55%       1.08s 23.68%  fmt.(*pp).printArg
      0.17s  3.73% 37.28%       0.17s  3.73%  runtime.pageIndexOf (inline)
      0.14s  3.07% 40.35%       0.28s  6.14%  fmt.(*buffer).writeString
(partial-inline)
      0.12s  2.63% 42.98%       0.15s  3.29%  runtime.findObject
      0.12s  2.63% 45.61%       0.12s  2.63%  runtime.futex
      0.12s  2.63% 48.25%       1.22s 26.75%  runtime.gcDrain
      0.11s  2.41% 50.66%       0.48s 10.53%  runtime.greyobject
      0.11s  2.41% 53.07%       2.38s 52.19%  vslc/src/backend/arm.genFunction
      0.10s  2.19% 55.26%       0.35s  7.68%  fmt.(*pp).handleMethods
      0.09s  1.97% 57.24%       0.14s  3.07%  strconv.appendEscapedRune
      0.09s  1.97% 59.21%       0.23s  5.04%  strconv.appendQuotedWith
      0.07s  1.54% 60.75%       0.07s  1.54%  runtime.(*itabTableType).find
      0.06s  1.32% 62.06%       0.06s  1.32%  runtime.nextFreeFast (inline)
      0.06s  1.32% 63.38%       0.06s  1.32%  runtime.releasem (inline)
      0.06s  1.32% 64.69%       0.10s  2.19%  sync.(*Pool).pin
      0.05s  1.10% 65.79%       0.05s  1.10%  runtime.heapBitsForAddr (inline)
      0.04s  0.88% 66.67%       0.05s  1.10%  fmt.(*pp).catchPanic
      0.04s  0.88% 67.54%       0.06s  1.32%  runtime.(*sweepLocked).sweep
      0.04s  0.88% 68.42%       0.17s  3.73%  runtime.gcWriteBarrier
      0.04s  0.88% 69.30%       0.04s  0.88%  runtime.heapBits.bits (inline)
      0.04s  0.88% 70.18%       0.07s  1.54%  runtime.heapBitsSetType
      0.04s  0.88% 71.05%       0.04s  0.88%  runtime.procPin (inline)
      0.04s  0.88% 71.93%       0.13s  2.85%  runtime.wbBufFlush1
      0.03s  0.66% 72.59%       0.05s  1.10%  fmt.(*buffer).write (inline)
      0.03s  0.66% 73.25%       0.46s 10.09%  fmt.(*pp).fmtString
```

Assembler generation -t = 4

```
Showing nodes accounting for 4.34s, 70.11% of 6.19s total
Dropped 165 nodes (cum <= 0.03s)
Showing top 30 nodes out of 131
      flat  flat%   sum%        cum   cum%
      0.40s  6.46%  6.46%       0.87s 14.05%  runtime.mallocgc
      0.37s  5.98% 12.44%       1.65s 26.66%  fmt.(*pp).doPrintf
      0.37s  5.98% 18.42%       0.37s  5.98%  runtime.markBits.isMarked (inline)
      0.36s  5.82% 24.23%       1.48s 23.91%  runtime.scanobject
      0.24s  3.88% 28.11%       0.24s  3.88%  runtime.memmove
      0.23s  3.72% 31.83%       0.33s  5.33%  runtime.findObject
      0.17s  2.75% 34.57%       0.19s  3.07%  runtime.pageIndexOf (inline)
      0.16s  2.58% 37.16%       0.33s  5.33%  fmt.(*buffer).writeString (inline)
      0.14s  2.26% 39.42%       0.14s  2.26%  runtime.futex
      0.14s  2.26% 41.68%       1.81s 29.24%  runtime.gcDrain
      0.14s  2.26% 43.94%       3.06s 49.43%  vslc/src/backend/arm.genFunction
      0.13s  2.10% 46.04%       1.02s 16.48%  fmt.(*pp).printArg
      0.12s  1.94% 47.98%       0.12s  1.94%  runtime.madvise
      0.12s  1.94% 49.92%       0.18s  2.91%  strconv.appendEscapedRune
      0.10s  1.62% 51.53%       0.12s  1.94%  runtime.heapBits.next (inline)
      0.09s  1.45% 52.99%       0.31s  5.01%  fmt.(*pp).handleMethods
      0.09s  1.45% 54.44%       0.16s  2.58%  runtime.getitab
      0.09s  1.45% 55.90%       0.09s  1.45%  runtime.nextFreeFast (inline)
      0.09s  1.45% 57.35%       0.27s  4.36%  strconv.appendQuotedWith
      0.09s  1.45% 58.80%       0.21s  3.39%  sync.(*Pool).Get
      0.08s  1.29% 60.10%       0.08s  1.29%  runtime.(*lfstack).pop (inline)
      0.08s  1.29% 61.39%       0.18s  2.91%  runtime.gcWriteBarrier
      0.08s  1.29% 62.68%       0.64s 10.34%  runtime.greyobject
      0.07s  1.13% 63.81%       0.10s  1.62%  fmt.(*fmt).fmtInteger
      0.07s  1.13% 64.94%       0.07s  1.13%  runtime.(*itabTableType).find
      0.07s  1.13% 66.07%       0.07s  1.13%  runtime.spanOf (inline)
      0.07s  1.13% 67.21%       0.14s  2.26%  runtime.wbBufFlush1
      0.06s  0.97% 68.17%       0.16s  2.58%  fmt.(*pp).free
      0.06s  0.97% 69.14%       0.06s  0.97%  runtime.heapBits.bits (inline)
      0.06s  0.97% 70.11%       0.09s  1.45%  runtime.heapBitsSetType
```

Assembler generation -t = 16

```
Showing nodes accounting for 6.37s, 68.13% of 9.35s total
Dropped 175 nodes (cum <= 0.05s)
Showing top 30 nodes out of 160
      flat  flat%   sum%        cum   cum%
      0.61s  6.52%  6.52%       2.13s 22.78%  runtime.scanobject
      0.50s  5.35% 11.87%       0.71s  7.59%  runtime.findObject
      0.48s  5.13% 17.01%       2.53s 27.06%  fmt.(*pp).doPrintf
      0.38s  4.06% 21.07%       0.38s  4.06%  runtime.markBits.isMarked (inline)
      0.36s  3.85% 24.92%       1.22s 13.05%  runtime.mallocgc
      0.35s  3.74% 28.66%       2.76s 29.52%  runtime.gcDrain
      0.29s  3.10% 31.76%       0.29s  3.10%  runtime.futex
      0.27s  2.89% 34.65%       0.27s  2.89%  runtime.pageIndexOf (inline)
      0.23s  2.46% 37.11%       0.50s  5.35%  fmt.(*buffer).writeString (inline)
      0.23s  2.46% 39.57%       0.23s  2.46%  runtime.madvise
      0.22s  2.35% 41.93%       1.76s 18.82%  fmt.(*pp).printArg
      0.22s  2.35% 44.28%       0.22s  2.35%  runtime.memmove
      0.19s  2.03% 46.31%       0.68s  7.27%  runtime.gcWriteBarrier
      0.19s  2.03% 48.34%       4.64s 49.63%  vslc/src/backend/arm.genFunction
      0.18s  1.93% 50.27%       0.18s  1.93%  runtime.(*lfstack).pop (inline)
      0.18s  1.93% 52.19%       0.18s  1.93%  runtime.nextFreeFast (inline)
      0.17s  1.82% 54.01%       0.86s  9.20%  runtime.greyobject
      0.15s  1.60% 55.61%       0.22s  2.35%  fmt.(*fmt).fmtInteger
      0.13s  1.39% 57.01%       0.51s  5.45%  fmt.(*pp).handleMethods
      0.13s  1.39% 58.40%       0.13s  1.39%  runtime.heapBits.bits (inline)
      0.12s  1.28% 59.68%       0.13s  1.39%  runtime.spanOf (inline)
      0.11s  1.18% 60.86%       0.11s  1.18%  runtime.memclrNoHeapPointers
      0.11s  1.18% 62.03%       0.49s  5.24%  runtime.wbBufFlush1
      0.10s  1.07% 63.10%       0.26s  2.78%  strconv.appendQuotedWith
      0.09s  0.96% 64.06%       0.16s  1.71%  strconv.appendEscapedRune
      0.08s  0.86% 64.92%       0.33s  3.53%  fmt.(*fmt).fmtS
      0.08s  0.86% 65.78%       3.44s 36.79%  fmt.Sprintf
      0.08s  0.86% 66.63%       0.16s  1.71%  runtime.getitab
      0.07s  0.75% 67.38%       0.34s  3.64%  fmt.(*pp).free
      0.07s  0.75% 68.13%       0.08s  0.86%  runtime.(*itabTableType).find
```

Appendix F: Profiling data memory

Memory profiling data, from *aamanyfuncs.vsl*, pulled from running *BenchmarkLIRGeneration* and *BenchmarkAssemblerGeneration* on the vm callsign system, with one, two, four and 16 worker goroutines using the vslc *-t* flag. Uninteresting nodes are cut automatically by the pprof *top* command, because their cumulative memory usage is very low.

LIR generation -t = 1

```
Showing nodes accounting for 278.89MB, 95.49% of 292.07MB total
Dropped 34 nodes (cum <= 1.46MB)
Showing top 10 nodes out of 43
      flat  flat%   sum%      cum   cum%
  104.62MB 35.82% 35.82%  104.62MB 35.82%  vslc/src/ir/lir.(*Block).CreateLoad
   99.24MB 33.98% 69.80%   99.24MB 33.98%  vslc/src/ir/lir.(*Block).CreateStore
      20MB  6.85% 76.65%        20MB  6.85%
vslc/src/ir/lir.(*Block).CreateFunctionCall
   15.51MB  5.31% 81.96%  16.51MB  5.65%
vslc/src/ir/lir.(*Block).CreateConstantInt
   15.50MB  5.31% 87.27%  15.50MB  5.31%
vslc/src/ir/lir.(*Block).createArithmeticInstruction
    8.01MB  2.74% 90.01%   273.88MB 93.77%  vslc/src/ir/lir.gen
    5.50MB  1.88% 91.89%   115.11MB 39.41%  vslc/src/ir/lir.genExpression
       5MB  1.71% 93.60%         5MB  1.71%  runtime.allocm
       3MB  1.03% 94.63%         3MB  1.03%
vslc/src/ir/lir.(*Function).CreateBlock
    2.50MB  0.86% 95.49%  2.50MB  0.86%  vslc/src/ir/lir.(*Module).CreateFunction
```

LIR generation -t = 2

```
Showing nodes accounting for 390.04MB, 95.66% of 407.72MB total
Dropped 46 nodes (cum <= 2.04MB)
Showing top 10 nodes out of 35
      flat  flat%   sum%        cum   cum%
  149.16MB 36.58% 36.58%   149.16MB 36.58%  vslc/src/ir/lir.(*Block).CreateLoad
  129.33MB 31.72% 68.31%   129.33MB 31.72%  vslc/src/ir/lir.(*Block).CreateStore
      30MB  7.36% 75.66%        30MB  7.36%
vslc/src/ir/lir.(*Block).CreateFunctionCall
   24.52MB  6.01% 81.68%  25.52MB  6.26%
vslc/src/ir/lir.(*Block).CreateConstantInt
   17.50MB  4.29% 85.97%  17.50MB  4.29%
vslc/src/ir/lir.(*Block).createArithmeticInstruction
   16.52MB  4.05% 90.02%   382.03MB 93.70%  vslc/src/ir/lir.gen
       7MB  1.72% 91.74%   157.65MB 38.67%  vslc/src/ir/lir.genExpression
    6.01MB  1.47% 93.21%  6.01MB  1.47%  runtime.allocm
    5.50MB  1.35% 94.56%  5.50MB  1.35%  vslc/src/ir/lir.(*Function).CreateBlock
    4.50MB  1.10% 95.66%  4.50MB  1.10%  vslc/src/ir/lir.(*Block).CreateDeclare
```

LIR generation -t = 4

```
Showing nodes accounting for 341.02MB, 95.02% of 358.87MB total
Dropped 46 nodes (cum <= 1.79MB)
Showing top 10 nodes out of 38
      flat  flat%   sum%        cum   cum%
  129.67MB 36.13% 36.13%   129.67MB 36.13%  vslc/src/ir/lir.(*Block).CreateLoad
  110.81MB 30.88% 67.01%   110.81MB 30.88%  vslc/src/ir/lir.(*Block).CreateStore
```

```
      26MB  7.25% 74.25%        26MB  7.25%
vslc/src/ir/lir.(*Block).CreateFunctionCall
   23.01MB  6.41% 80.67%  24.01MB  6.69%
vslc/src/ir/lir.(*Block).CreateConstantInt
      19MB  5.29% 85.96%        19MB  5.29%
vslc/src/ir/lir.(*Block).createArithmeticInstruction
   15.02MB  4.18% 90.15%   335.01MB 93.35%  vslc/src/ir/lir.gen
    5.50MB  1.53% 91.68%  5.50MB  1.53%  vslc/src/ir/lir.(*Module).CreateFunction
    5.50MB  1.53% 93.21%   139.15MB 38.77%  vslc/src/ir/lir.genExpression
       4MB  1.12% 94.33%         4MB  1.12%  runtime.allocm
    2.50MB   0.7% 95.02%  2.50MB   0.7%  vslc/src/ir/lir.(*Block).CreateDeclare
```

LIR generation -t = 16

```
Showing nodes accounting for 360.07MB, 96.20% of 374.30MB total
Dropped 50 nodes (cum <= 1.87MB)
Showing top 10 nodes out of 33
      flat  flat%   sum%        cum   cum%
  138.67MB 37.05% 37.05%   138.67MB 37.05%  vslc/src/ir/lir.(*Block).CreateLoad
  127.36MB 34.03% 71.07%   127.36MB 34.03%  vslc/src/ir/lir.(*Block).CreateStore
   25.51MB  6.82% 77.89%  26.01MB  6.95%
vslc/src/ir/lir.(*Block).CreateConstantInt
      22MB  5.88% 83.77%        22MB  5.88%
vslc/src/ir/lir.(*Block).CreateFunctionCall
      16MB  4.27% 88.04%        16MB  4.27%
vslc/src/ir/lir.(*Block).createArithmeticInstruction
   13.51MB  3.61% 91.65%   352.56MB 94.19%  vslc/src/ir/lir.gen
      5.50MB  1.47% 93.12%      6MB  1.60%
vslc/src/ir/lir.(*Module).CreateFunction
      4.50MB  1.20% 94.33%   141.15MB 37.71%  vslc/src/ir/lir.genExpression
      4MB  1.07% 95.40%         4MB  1.07%  runtime.allocm
      3MB   0.8% 96.20%         3MB   0.8%
vslc/src/ir/lir.(*Block).CreateDeclare
```

Assembler generation -t = 1

```
Showing nodes accounting for 214.56MB, 95.44% of 224.82MB total
Dropped 35 nodes (cum <= 1.12MB)
Showing top 10 nodes out of 46
      flat  flat%   sum%        cum   cum%
  121.55MB 54.07% 54.07%   121.55MB 54.07%  strings.(*Builder).WriteString
   30.50MB 13.57% 67.63%   181.35MB 80.66%  vslc/src/backend/arm.genFunction
      28MB 12.45% 80.09%        29MB 12.90%  fmt.Sprintf
   15.50MB  6.90% 86.98%  15.50MB  6.90%  vslc/src/backend/arm.CreateRegisterFile
      5MB  2.23% 89.21%         5MB  2.23%  runtime.allocm
      4.50MB  2.00% 91.21%      6.50MB  2.89%
vslc/src/backend/arm.genFunctionCall
      4MB  1.78% 92.99%         6MB  2.67%  vslc/src/backend/arm.genExpression
      2.50MB  1.11% 94.10%   211.06MB 93.88%  vslc/src/backend/arm.GenArm
      1.50MB  0.67% 94.77%      1.50MB  0.67%  vslc/src/frontend.nodeInit
      1.50MB  0.67% 95.44%      2MB  0.89%  vslc/src/ir/lir.(*String).Name
```

Assembler generation -t = 2

```
Showing nodes accounting for 295.92MB, 96.81% of 305.68MB total
Dropped 33 nodes (cum <= 1.53MB)
Showing top 10 nodes out of 34
      flat  flat%   sum%        cum   cum%
  161.91MB 52.97% 52.97%   161.91MB 52.97%  strings.(*Builder).WriteString
   52.50MB 17.18% 70.14%   53.50MB 17.50%  fmt.Sprintf
      36MB 11.78% 81.92%   243.42MB 79.63%  vslc/src/backend/arm.genFunction
      24MB  7.85% 89.77%        24MB  7.85%
vslc/src/backend/arm.CreateRegisterFile
      7MB  2.29% 92.06%   32.73MB 10.71%  vslc/src/backend/arm.genFunctionCall
      5MB  1.64% 93.70%   52.22MB 17.08%  vslc/src/backend/arm.genExpression
      4.50MB  1.47% 95.17%      4.50MB  1.47%  runtime.allocm
      2.50MB  0.82% 95.99%      49.50MB 16.19%  vslc/src/backend/arm.GenArm
      1.50MB  0.49% 96.48%      4MB  1.31%  vslc/src/ir/lir.(*String).Name
      1MB  0.33% 96.81%   5.51MB  1.80%  vslc/src/util.(*Writer).Label
```

Assembler generation -t = 4

```
Showing nodes accounting for 319.55MB, 97.65% of 327.25MB total
Dropped 53 nodes (cum <= 1.64MB)
Showing top 10 nodes out of 29
      flat  flat%   sum%        cum   cum%
  193.04MB 58.99% 58.99%   193.04MB 58.99%  strings.(*Builder).WriteString
   45.50MB 13.90% 72.89%  45.50MB 13.90%  fmt.Sprintf
      32MB  9.78% 82.67%   249.72MB 76.31%  vslc/src/backend/arm.genFunction
      27MB  8.25% 90.92%        27MB  8.25%
vslc/src/backend/arm.CreateRegisterFile (inline)
      5.50MB  1.68% 92.61%      21.09MB  6.44%
vslc/src/backend/arm.genExpression
      5MB  1.53% 94.13%         5MB  1.53%  runtime.allocm
      4MB  1.22% 95.36%   8.01MB  2.45%  vslc/src/util.(*Writer).Label
      3MB  0.92% 96.27%   64.32MB 19.66%  vslc/src/backend/arm.GenArm
      3MB  0.92% 97.19%   17.54MB  5.36%  vslc/src/backend/arm.genFunctionCall
      1.50MB  0.46% 97.65%      5MB  1.53%  vslc/src/ir/lir.(*String).Name
```

Assembler generation -t = 16

```
Showing nodes accounting for 328.94MB, 96.84% of 339.66MB total
Dropped 32 nodes (cum <= 1.70MB)
Showing top 10 nodes out of 37
      flat  flat%   sum%        cum   cum%
  171.43MB 50.47% 50.47%   171.43MB 50.47%  strings.(*Builder).WriteString
   63.50MB 18.70% 69.17%        67MB 19.73%  fmt.Sprintf
   46.50MB 13.69% 82.86%   249.04MB 73.32%  vslc/src/backend/arm.genFunction
      27MB  7.95% 90.81%        27MB  7.95%
vslc/src/backend/arm.CreateRegisterFile
      6MB  1.77% 92.57%   27.08MB  7.97%  vslc/src/backend/arm.genExpression
      3.50MB  1.03% 93.60%      4MB  1.18%  runtime.allocm
      3MB  0.88% 94.49%         3MB  0.88%  runtime.malg
      3MB  0.88% 95.37%   26.56MB  7.82%  vslc/src/backend/arm.genFunctionCall
      3MB  0.88% 96.25%   10.57MB  3.11%  vslc/src/util.(*Writer).Label
      2MB  0.59% 96.84%         2MB  0.59%  fmt.glob..func
```

Appendix G: Buffer example code

This example code appends integers and a string literal to a slice. The length of the slice is printed when the slice is re-allocated.

```
package main

import "fmt"

func main() {
    fmt.Println("Testing int buffer")
    arr := make([]int, 0)
    for i1 := 0; i1 < 256; i1++ {
       if len(arr) == cap(arr) {
             fmt.Printf("cap: %d\n", cap(arr))
       }
       arr = append(arr, i1)
    }

    fmt.Println("\nTesting string buffer")
    str := "this is a test"
    strarr := make([]byte, 0)
    for i1 := 0; i1 < 256; i1++ {
       if cap(strarr)-len(strarr) < len(str) {
             fmt.Printf("cap: %d\n", cap(strarr))
       }
       strarr = append(strarr, str...)
    }
}

// Which produces output:
//
// Testing int buffer
// cap: 0
// cap: 1
// cap: 2
// cap: 4
// cap: 8
// cap: 16
// cap: 32
// cap: 64
// cap: 128
//
// Testing string buffer
// cap: 0
// cap: 16
// cap: 32
// cap: 64
// cap: 128
// cap: 256
// cap: 512
// cap: 896
// cap: 1408
// cap: 2048
// cap: 3072
```

Appendix H: Append example code

This example code appends a string literal to one or multiple buffers/slices. The number of times the buffers are re-allocated is counted in the *calcAppends* function. Loop variable *i1* defines how many times the string literal will be appended to the *t* buffers. Each buffer holds *i1 / t* string literals.

```
package main

import "fmt"

// str defines the string to append to the buffer.
var str = "This is an example string"

// calcAppends appends str to the buffer buf n times.
func calcAppends(buf []byte, n int) (a int) {
    for i1 := 0; i1 < n; i1++ {
        if cap(buf)-len(buf) < len(str) {
            // buffer can't hold string, it will reallocate a new buffer
            a++
        }
        buf = append(buf, str...)
    }
    return a
}

func main() {
    // routines define the number of parallel worker go routines.
    routines := []int{1, 2, 4, 16}

    // adjust how many times to append string to buffer.
    // i1 is chosen to be easily divisible by all routine numbers.
    for i1 := 16; i1 <= 1024; i1 <<= 1 {
        // sequential, one big buffer.
        fmt.Printf("Appending %d times\n", i1)

        for _, t := range routines {
            job := i1 / t // number of times to append for a worker.
            m := 0
            for i2 := 0; i2 < t; i2++ {
                buf := make([]byte, 0)
                m += calcAppends(buf, job)
            }
            fmt.Printf("t = %d: allocations %d\n", t, m)
        }
        fmt.Println("")
    }
}

// Which produces output:
//
// Appending 16 times
// t = 1: allocations 5
// t = 2: allocations 8
// t = 4: allocations 12
// t = 16: allocations 16
//
```

```
// Appending 32 times
// t = 1: allocations 6
// t = 2: allocations 10
// t = 4: allocations 16
// t = 16: allocations 32
//
// Appending 64 times
// t = 1: allocations 8
// t = 2: allocations 12
// t = 4: allocations 20
// t = 16: allocations 48
//
// Appending 128 times
// t = 1: allocations 10
// t = 2: allocations 16
// t = 4: allocations 24
// t = 16: allocations 64
//
// Appending 256 times
// t = 1: allocations 12
// t = 2: allocations 20
// t = 4: allocations 32
// t = 16: allocations 80
//
// Appending 512 times
// t = 1: allocations 15
// t = 2: allocations 24
// t = 4: allocations 40
// t = 16: allocations 96
//
// Appending 1024 times
// t = 1: allocations 17
// t = 2: allocations 30
// t = 4: allocations 48
// t = 16: allocations 128
```