Jason D. Davies

Optimising Lua

Computer Science Tripos Part II

St John's College

2005

Proforma

Name:	**Jason D. Davies**
Project title:	**Optimising Lua**
Examination:	**Computer Science Tripos Part II, 2005**
Word count:	**11,777**[1]
Project originator:	**E. C. Upton**
Project supervisor:	**E. C. Upton**

Original Aims of the Project

Originally, I aimed to implement a just-in-time compiler for the Lua programming language, in order to considerably improve run-time performance. The compiler would be consistent with the operation of the standard Lua interpreter and correctly evaluate a set of test cases. The run-time performance improvement would be evaluated using benchmarks.

Work Completed

I have successfully developed an optimising just-in-time compiler for Lua, which resulted in considerable performance improvements as shown by benchmarks. Since Lua is dynamically-typed, data-flow analysis of types of values is performed in order to optimise the compiled code further. All regression test cases were passed, showing the consistency of the implementation with the standard Lua interpreter.

Special Difficulties

None.

[1] Word count approximated using `detex dissertation.tex | wc -w`

Declaration of Originality

I, Jason D. Davies of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Lua is an embeddable scripting language originally developed at Tecgraf, PUC-Rio in 1993 and now widely used in the interactive entertainment industry[1]. Its main goals are to be simple, efficient, portable and lightweight. The objective of this project was to improve the performance of the Lua programming language interpreter by adding a just-in-time (JIT) compiler to the standard run-time environment.

Although it is among the fastest scripting languages, Lua is still approximately a factor of 20 slower than C in the majority of benchmarks [**?**, **?**]. There is still plenty of room for improvement. Admittedly, the performance of the standard Lua virtual machine (VM) is adequate for tasks such as configuration or for use as a 'glue' language, but high performance and efficiency are important in games and interactive applications.

Compiled native machine code usually executes significantly faster than interpreted code; there is no interpretive overhead of decoding and dispatching individual bytecode operations. This is the primary motivation for writing a JIT compiler; even a naïve straight translation from intermediate bytecode into native machine code would eliminate this interpretive overhead and result in a performance increase.

Traditionally, most JIT compilers have been written for statically-typed languages, such as Java. Lua is *dynamically-typed*, however, and merely removing the interpretive overhead would fall short of the expected performance increase due to the additional overheads of checking and copying the types of values at run-time. Thus Lua presents an interesting challenge: to obtain a more significant performance increase by eliminating redundant type checks, thereby closing the performance gap between statically and dynamically-typed languages.

1.1 Overview of Lua

The standard Lua distribution includes a standalone command-line interpreter, but Lua is designed primarily to be embedded into a 'host' application. The host application controls when a script is loaded, compiled and executed and catches any errors raised during the execution of a script. A script's capabilities can also be extended by making additional libraries available from the host application. For

[1] An informal poll conducted in September 2003 by gamedev.net, an important site for game programmers, showed Lua as the most popular scripting language for game development.

```
1  function newton(a, x)
2      local tolerance = 1e-10
3      local oldx
4      repeat
5          oldx = x
6          x = x - (x^2 - a) / (2 * x - 0)
7      until math.abs(x - oldx) < tolerance
8      return x
9  end
10
11 print(newton(2, 1))
```

Figure 1.1: Example: Newton's method in Lua

these reasons, Lua is known as an 'embedded extension language.'

The standalone interpreter is useful for development and debugging; it can be run in an interactive mode, where each line typed executes immediately after being entered.

The Lua runtime environment contains a fast static compiler, which generates intermediate bytecode for an abstract register machine. This bytecode is subsequently interpreted by an interpretive VM. As with Java, Lua scripts can be precompiled into the intermediate bytecode format, saved to disc and executed later. Scripts seldom need to be precompiled, however, since the static compiler is extremely fast.

For a complete history of Lua's development, see [**?**].

1.1.1 Syntax and Semantics

Lua is syntactically similar to Pascal and semantically similar to Scheme. Figure 1.1 shows an example Lua program, which calculates the square root of 2 using Newton's method.

On line 1 the `newton` function is defined, taking 2 arguments a and x. Since Lua is dynamically-typed, no types are specified in the argument definitions. Lines 2 and 3 declare local variables using the `local` keyword. The scope of these variables is limited to the `newton` function. Global variables need no declaration and can be accessed from any scope in the current Lua context. Lines 4 and 7 define a loop which repeatedly executes its body, lines 5 and 6, until the expression `math.abs(x - oldx) < tolerance` is true. Finally, the result is returned using the `return` keyword on line 8. Line 11 illustrates how the `newton` function can be called, and prints the result to standard output.

1.1.2 Dynamic Typing

Lua is a dynamically-typed language, which means it has no type declarations. Types are associated with values, not variables as in statically-typed languages; each variable in Lua stores a pair (t, v), where t is an integer tag identifying the type of the value v.

Figure 1.2 shows how Lua values are represented in C. The `TObject` structure represents the tagged union (t, v), with the `tt` field being the type tag t and `value` being the value v. The `Value` union represents values. The `n` field is used for numbers, `b` for booleans, `p` for light userdata and `gc` for other values (strings, tables,

```
typedef struct {
    int tt;             /* type tag */
    Value value;        /* union value */
} TObject;

typedef union {
    GCObject *gc;
    void *p;
    lua_Number n;       /* 8 bytes if double-precision is used */
    int b;
} Value;
```

Figure 1.2: Lua values are tagged unions

Type	Description
`nil`	A marker type representing the single value `nil`.
`boolean`	Has a value of either `true` or `false`.
`number`	Represents a double-precision floating-point number[2].
`string`	Byte arrays, which are explicitly sized so can contain embedded zeroes.
`table`	Associative arrays, which can be indexed by any value except `nil`.
`function`	Either Lua or C functions.
`userdata`	Can hold arbitrary C data.
`thread`	Represents an independent thread of execution, used for coroutines.

Table 1.1: Lua's data types

functions, heavy userdata and threads), which are subject to garbage collection. The built-in polymorphic function `type` : $\alpha \rightarrow$ `string` can be used at runtime to retrieve a description of the type of a value.

Lua defines only 8 data types in total, summarised in Table 1.1.

Coercion of Types

String and number types are automatically converted between each other: whenever a string is used in a context where a number is expected, the string is converted into a number, and vice versa. An error is raised if a string cannot be converted into a number, as demonstrated in the interactive Lua session in Figure 1.3.

This behaviour must be retained by the JIT compiler to ensure consistent execution of arbitrary Lua programs.

1.1.3 Tables

The table is the only structured data type available in Lua; a table is an associative map, an abstract data type similar to an array but which can be indexed by any value (except `nil`). Commonly used data structures such as trees, lists and graphs can be represented using tables. The code in Figure 1.4 reads lines from standard input, stores them in a linked list and then prints them out in reverse order.

[2]The numerical representation used can be changed at compile-time e.g. C's single-precision *float* or integer representations *int* or *long*.

```
> = 100 + "7"
107
> = "1000" + 234
1234
> = "hello " + 1234
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
        stdin:1: in main chunk
        [C]: ?
> = "hello " .. 1234
hello 1234
```

Figure 1.3: Demonstrating coercion of `string` and `number` types.

```
1  list = nil
2  for line in io.lines() do
3      list = {next=list, value=line}
4  end
5
6  l = list
7  while l do
8      print(l.value)
9      l = l.next
10 end
```

Figure 1.4: Implementing linked lists using Lua's table type.

Tables are used extensively by the vast majority of programs and so the Lua developers have spent much effort on making them efficient.

1.1.4 Metamethods

Another powerful concept in Lua is the alteration of the semantics for userdata and table accesses through using *metamethods*. Metamethods allow programs to overload certain operations on tables and userdata objects. Each overloaded object has a *metatable* of functions (metamethods) associated with it. The metamethods are associated with events, which occur when operations such as addition and comparisons are performed on the object. This can even be used to implement object-oriented mechanisms [**?**].

1.1.5 Libraries

The standard Lua distribution includes a set of core libraries for performing I/O, math and string operations; it is straightforward to add extra user-defined libraries, written either in C or Lua. These extra libraries can be dynamically loaded at runtime.

It is interesting to note that the majority of Java's standard libraries are written in Java, whereas Lua's standard libraries are mainly written in C for speed. With the addition of the JIT compiler, it may be feasible to rewrite many of Lua's standard libraries in Lua.

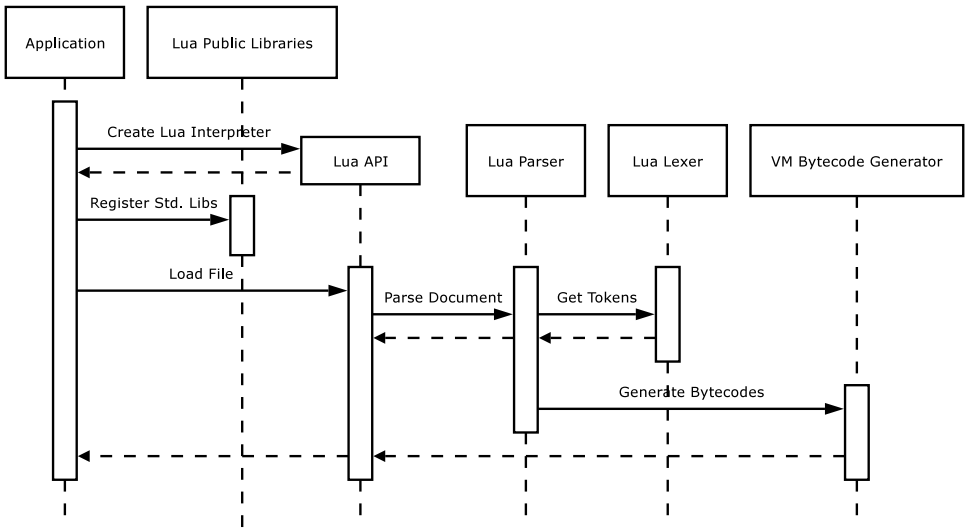

Figure 1.5: The process of initialising the standard Lua VM and running a program.

1.1.6 Coroutines

Coroutines are similar to threads in that they each have their own stack, local variables and instruction pointer. Unlike threads, however, only one coroutine is ever executing at a given time, explicitly suspending its execution itself when necessary. Lua supports asymmetric coroutines [**?**]. Coroutines are also supported by languages such as BCPL [**?**] and Simula [**?**].

1.1.7 Architecture

Figure 1.5 gives an overview of Lua's execution model. First of all the application creates a Lua interpreter state, which is required by the Lua C API. Next, all extension libraries are registered. Then scripts can be loaded, parsed and translated into bytecode. Finally, the bytecode is executed by the VM.

1.2 Just-In-Time Compilation

A common way to improve the execution speed of interpreted languages is *just-in-time* or *dynamic* compilation. The overhead of decoding the source language is removed by translating it to native machine code on-demand at run-time and the result is cached for further use. For statically-typed languages, this approach can improve speed by an order of magnitude over interpretive execution. Modern JIT compilers also apply traditional compiler optimisations such as software pipelining and low-level register allocation.

1.2.1 Drawbacks

There are several trade-offs to consider when using the technique of just-in-time compilation. The initial translation to native machine code is often costly in terms of time and this must be offset by the speedup gained by running the native code. Unless the performance gain is large, it may not be worth translating into machine

code. In *Mixed-mode Bytecode Execution* [**?**], Agesen and Detlefs show how the combination of interpreting and compiling modes yields high performance.

The VM will typically oscillate between the JIT compiler's main bytecode translation loop and executing a compiled native code buffer. It is possible that this will thrash the instruction cache, particularly if the native code buffer is large.

The speed increase is generally only available at the cost of higher memory usage. The Lua bytecodes are very compact but each opcode can generate over 20 native code instructions, increasing memory usage by an order of magnitude. In comparison, several Java VM instructions often translate into a single native code instruction. This is significant in embedded systems where memory constraints are tight.

There is a tension between compilation speed and code quality. Traditional offline compilers are free to spend more time performing larger scale global optimisations, whereas a JIT compiler cannot afford to spend too much time on compilation. Thus JIT compilers tend to perform only cheap local optimisations.

1.3 Existing Work

An ongoing goal of the Lua developers has been to increase the performance of the virtual machine. The release of version 5.0 saw the original stack-based virtual machine replaced with a register-based virtual machine. Register-based code avoids the need for 'push' and 'pop' instructions to move values around on the stack, which are particularly expensive in Lua since they involve copying tagged values, as we shall see in Section 2.2.2. Tables were also optimised for the (common) case where they are used as arrays with integer indices. Overall, these improvements resulted a speed increase of up to 40% relative to version 4.0 [**?**].

Portability is one of Lua's primary concerns, and performance may have been sacrificed in some areas because of this. There is some evidence to suggest that the performance of bytecode language interpreters can be improved dramatically using dynamic binary translation [**?**], but this is not very portable.

To my knowledge, this is the first JIT compiler ever written for Lua. Java already has several. One reason for the lack of a Lua JIT compiler might be that Lua is designed to be very lightweight and portable across different platforms.

There is a large body of reference material available on JIT compilers written for other languages. One of the earliest examples is ORBIT [**?**], an optimising compiler for Scheme, which uses trace-scheduling.

Python is very similar to Lua in that it is also dynamically-typed and has a clean syntax. Work has been done recently on a specialising JIT compiler known as *Psyco* [**?**, **?**]. Unlike traditional JIT compilers, Psyco can use run-time data to potentially generate several versions of the native code, each specialised for values of different types.

Another related piece of work is *Pirate* [**?**], a compiler for Lua targeting the Parrot virtual machine. Parrot has been designed to support the dynamically-typed language Perl, but can be used to execute other dynamically-typed languages too. It includes a JIT compilation subsystem.

Chapter 2

Preparation

Since this was an optimisation project, I was aware that once the basic just-in-time (JIT) compiler had been implemented, any number of further optimisation phases could be added as time permitted. However, preliminary research was necessary before embarking on the implementation itself. I was unfamiliar with the internals of the Lua VM, and I had not done much low-level programming in C or assembly language before.

I read Roberto Ierusalimschy's book, Programming in Lua [**?**] and found it very informative and useful for familiarising myself with the Lua language. The Lua community [**?**] also has a number of resources, including a mailing list [**?**] and an online collaborative Wiki [**?**] containing documentation and sample code.

I spent some time researching compiler optimisation techniques in order to design the overall system. Some excellent references for this were Steven Muchnick's *Advanced Compiler Design & Implementation* [**?**] and the somewhat dated but still relevant *Compilers: Principles, Techniques and Tools* [**?**].

The full source code for the standard Lua interpreter is freely available under the terms of the MIT license; I was able to reuse the code for the lexer, parser, VM and standard Lua libraries as a basis for the JIT compiler. The source code for version 5.1 is just over 16,000 lines long in total; this is considerably shorter than other comparable language VMs such as Python, which is over 50,000 lines long.

One of my concerns when starting this project was whether I would be able to understand the internal workings of Lua's VM in the time available. The Lua core turned out to be mostly well-written, although there were few comments in the code and it made heavy use of C macros. I was able to investigate how the VM worked in great detail by stepping through its execution using the *gdb* debugger.

I decided to follow an incremental development model [**?**] as closely as possible during the design and implementation of the project, new optimisation phases being added to the system as time permitted.

Most users of Lua do not need to know how the Lua VM is implemented so there is no official specification for the VM instruction set. I would base my work on a particular version of the standard Lua VM, version 5.1-work5, which is in current development with an alpha release expected soon. I chose to base my work on this version as it contains the most up-to-date optimisations to benchmark against.

2.1 Requirements Analysis

The following project requirements are based on the original success criteria set out in the project proposal, included in Appendix C.

1. The JIT compiler should produce results that are as consistent as possible with the standard Lua interpreter. This includes coercion of types and the metamethod mechanism, introduced in Section 1.1. Any inconsistencies should be clearly identified.

2. Optimisations should be performed on the intermediate Lua bytecode.

3. Heuristics should be used to determine whether to use the JIT compiler for a particular sequence of Lua bytecode.

4. The JIT compiler system should result in a reasonable performance increase when compared with the standard Lua interpreter. At this stage, it was difficult to quantify a required performance increase. See Section 4.3 for benchmark results.

5. The target machine architecture should be Intel x86 (i386-compatible and above).

6. The platform independent portions of the JIT compiler and other modules should be written in standards-compliant ANSI C, for greater portability between x86 operating systems and C compilers.

7. The project should be split into modules where possible, for easier testing, debugging and maintenance. All code should be clearly commented and formatted.

2.2 Specification

2.2.1 Supported Lua Instructions

Lua 5.1 currently has 36 opcodes in total, listed in Table 2.1 (modified from *A No-Frills Introduction to Lua VM Instructions* [**?**]).

It is unnecessary to support all 36 instructions; contiguous blocks of supported instructions would be compiled into native machine code by the JIT compiler, and the standard interpretive VM used for unsupported instructions.

Due to using an incremental development model, it was unnecessary to specify precisely which opcodes to support. However, it was possible to specify the order upon which they would be worked. To begin with, straight line (i.e. non-control-flow) instructions would be implemented, followed by control-flow instructions and more complex instructions if time was available:

1. **Data Instructions**
   Among the simplest Lua instructions are `MOV` and `LOADK`, which transfer data between local variables and load constant values into local variables respectively. `LOADNIL` is also straightforward to implement.

2. **Arithmetic and Logic Instructions**
   These are `ADD`, `SUB`, `MUL`, `DIV`, `POW`, `UNM` and `NOT`.

3. **Tables, Upvals and Global Variables**
   In the standard Lua VM, global variables are implemented using the ubiquitous table data structure, so the native machine code emitted for these instructions would be fairly similar. `GETGLOBAL`, `SETGLOBAL`, `GETTABLE`, `SETTABLE`, `GETUPVAL` and `SETUPVAL`.

4. **String Concatenation**
   Strings are concatenated using the `CONCAT` instruction.

5. **Control Flow Instructions**
   Any instructions that have more than one successor in the control flow graph. See Section **??** for more details on flow graphs. A control flow graph should be constructed before these instructions are compiled, so that:

   (a) if a successor is beyond an unsupported instruction in the Lua instruction stream, we cannot support the current instruction;

   (b) facilitates easier tracking of jump targets, especially forward jumps.

   The following Lua instructions fall into this category:

   - Unconditional jumps: `JMP`
   - Comparisons: `EQ`, `LT`, `LE`, `TEST`
   - Boolean copy with skip: `LOADBOOL`
   - Numerical and table loops: `FORPREP`, `FORLOOP`, `TFORPREP`, `TFORLOOP`

6. **Function Calls and Returns**
   Both C and Lua functions can be called from Lua. In order to preserve coroutine semantics, the C stack must be kept constant since `coroutine.yield()` assumes that the C stack is constant.

2.2.2 Areas of Optimisation

Lua's internal compiler is executed every time a Lua source file is loaded. In order to be as fast and compact as possible, it does not heavily optimise the intermediate bytecode it generates. Some additional inexpensive peephole optimisations could still be added, in particular *constant folding*, which refers to the evaluation at compile time of expressions whose operands are known to be constant. For instance, `a and b or c` is a common Lua idiom, used to simulate a ternary operator. When `b` is a constant, a redundant test on `b` is generated by the compiler.

These static compiler optimisations are outside the scope of this project, and could be investigated as a possible extension.

Switch Overhead

One method whereby JIT compilers for statically-typed languages typically get improvement is removing the switch overhead, i.e. the overhead of decoding and dispatching individual instructions. As explained in Section 1.2.1, however, the switch overhead is less significant in Lua because each instruction effectively does

Opcode	Name	Arguments	Description
0	MOVE	A B	Copy a value from one register to another
1	LOADK	A Bx	Load a constant into a register
2	LOADBOOL	A B C	Load a boolean into a register
3	LOADNIL	A B	Load `nil` into a range of registers
4	GETUPVAL	A B	Load an upvalue into a register
5	GETGLOBAL	A Bx	Load a global variable into a register
6	GETTABLE	A B C	Load a table element into a register
7	SETGLOBAL	A Bx	Store a register into a global variable
8	SETUPVAL	A B	Store a register into an upvalue
9	SETTABLE	A B C	Store a register into a table element
10	NEWTABLE	A B C	Create a new table
11	SELF	A B C	Prepare an object method for calling
12	ADD	A B C	Addition
13	SUB	A B C	Subtraction
14	MUL	A B C	Multiplication
15	DIV	A B C	Division
16	POW	A B C	Exponentiation
17	UNM	A B	Unary minus
18	NOT	A B	Logical NOT
19	CONCAT	A B C	Concatenate a range of registers containing strings
20	JMP	sBx	Unconditional jump
21	EQ	A B C	Equality test
22	LT	A B C	Less-than test
23	LE	A B C	Less-than-or-equal-to test
24	TEST	A B C	Test for short-circuit logical AND and OR evaluation
25	CALL	A B C	Call a function closure
26	TAILCALL	A B C	Perform a tail call
27	RETURN	A B	Return from a function call
28	FORLOOP	A sBx	Numeric for loop
39	FORPREP	A sBx	Initialise numeric for loop
30	TFORLOOP	A C	Generic for loop
31	TFORPREP	A sBx	Initialise generic for loop
32	SETLIST	A B C	Set a range of array elements for a table
33	CLOSE	A	Close a range of locals being used as upvalues
34	CLOSURE	A Bx	Create a closure of a function prototype
35	VARARG	A B	Create a variable argument function closure

Table 2.1: Summary of Lua instructions

```
1  case OP_ADD: {              /* handle the ADD instruction */
2    TValue *rb = RKB(i);      /* get values of operand B    */
3    TValue *rc = RKC(i);      /* get values of operand C    */
4    if (ttisnumber(rb) && ttisnumber(rc))
5      setnvalue(ra, num_add(nvalue(rb), nvalue(rc)));
6    else
7      base = Arith(L, ra, rb, rc, TM_ADD, pc);
8    continue;
9  }
```

Figure 2.1: Extract from the standard Lua interpretive VM demonstrating type checking.

more work. Other overheads concerning type checks also exist, which are restricted to dynamically-typed languages. Removing the switch overhead would produce a speedup but other possible areas for optimisation need to be investigated to obtain performance nearer to that of a JIT-compiled statically-typed language.

Type Checks

In practically every Lua instruction, the interpreter performs several type checks to ensure type safety and to determine which variant operation to execute; most Lua instructions will fall back to the metamethods (see Section 1.1.4) if the type of a value is not supported. The numerical instructions also convert between numerical and string types automatically. The typical operation of a Lua instruction is as follows:

1. Perform type checks on operand values.
2. Perform some operation on the values
3. Write the result to the destination, along with its type tag.

Figure 2.1 shows an extract from the standard Lua VM's main interpreter loop, illustrating how the types of `ADD`'s operands are checked before any arithmetic is performed (line 4). If both values are numbers, then they are added and the result is written to the destination register along with a numeric type tag (line 5); otherwise the more heavyweight `Arith()` function is called (line 6), which checks for handlers for those types to fall back to and throws an error if none are found.

If the types of values are known in advance, these type checks become redundant and can be eliminated completely. In addition, the type tag may not need to be written to the destination register if it already contains a value of the appropriate type, thus the number of memory accesses may be reduced.

Memory Accesses

Memory accesses are normally expensive, depending on whether data have been cached somewhere in the memory hierarchy. Optimisations here would focus on reducing the memory traffic. For example, copying of type tags can be reduced as discussed above; code could also be optimised to exploit caches better; and constant values may be injected directly into the instruction stream instead of being loaded from memory, so that the instruction cache may be exploited.

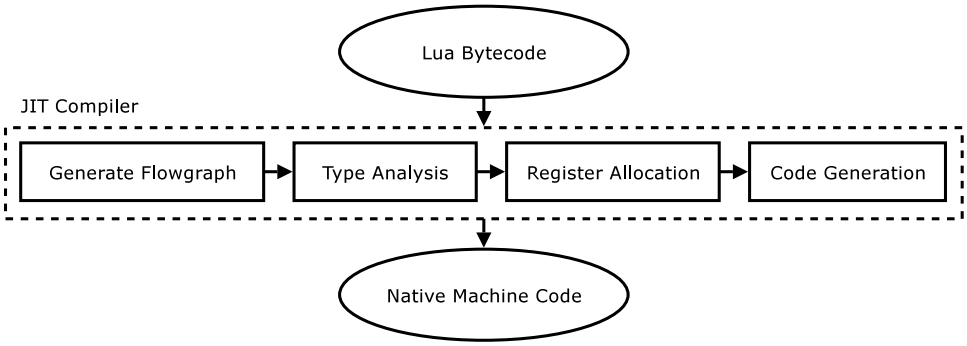

Figure 2.2: The modular design of the JIT compiler system.

The number of loads and stores to memory can also be reduced by caching values in the processor's register file, using a technique known as *register allocation*. There are a number of different register allocation algorithms and these are discussed in more detail in Section 2.2.3.

JIT Compiler Heuristics

Although the native code produced by the JIT compiler will usually execute faster, there is an initial overhead when compiling Lua instructions into native code. Typically, this means that a block of code which is only executed once will run slower when the JIT compiler is used.

The majority of programs spend most of their time executing a small fraction of their code. Therefore, instead of using the JIT compiler for all sequences of code, it is advantageous to only use it for those critical regions of code that are executed often.

No perfect oracle for predicting the future exists, but heuristics can be used to derive a reasonable approximation to the future using information from the past execution of the program. By running the program using the interpreter and analysing it as it runs, the critical regions can be estimated and the JIT compiler focused on these regions for maximum payoff. Sun's Hotspot engine [**?**] for Java effectively does this by identifying 'hotspots' that are likely to be executed the most.

Many programs consist of a region of startup code (and possibly shutdown code) which is only executed once. The program then spends the majority of its time in the 'main' region, which is repeatedly executed many times. The aim of the heuristics is to discriminate between these regions.

A very simple heuristic is this: only call the JIT compiler the second time a region of code is executed. Thus, startup code and shutdown code that is only executed once will not suffer the overhead of JIT compilation, but the main program will benefit from the JIT compilation and overcome its initial JIT compilation overhead.

2.2.3 Modular Design

After considering the requirements and the various areas of optimisation, the modular design shown in Figure 2.2 was used. Each module is described below:

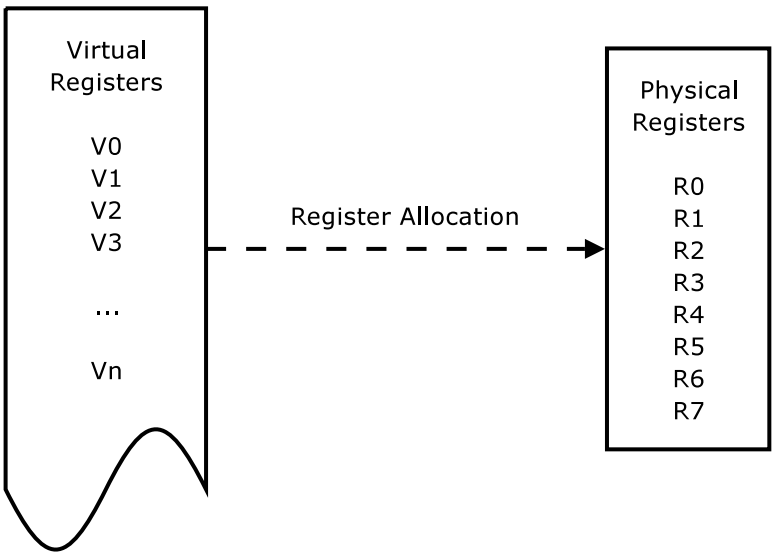

Figure 2.3: Register allocation: a mapping from virtual registers onto a finite number of physical registers.

JIT Compiler

This is the main module, which should be invoked from the Lua VM and passed a sequence of Lua code. The module should return either a function pointer to the compiled native machine code, or `NULL` if it was not deemed advantageous to perform the translation, in which case the Lua VM should fall back to an interpretive mode.

The compiled native code should be cached so that further calls to this module with the same Lua code sequence will not incur the unnecessary overhead of recompilation.

The JIT compiler heuristics for deciding whether to JIT-compile a given Lua bytecode sequence will also take place here.

Basic Blocks and Flow Graphs

A *basic block* is a straight-line sequence of code that can be entered only at the beginning and exited only at the end.

A graph representation of a program, known as a *flow graph*, is needed to perform type propagation data-flow analysis (described later in this section) and to implement the control-flow instructions. Nodes in the flow graph represent computations, usually represented by basic blocks, and the edges represent the flow of control in the program.

This module should separate a sequence of Lua instructions into basic blocks, and use them to construct a flow graph.

Register Allocation

The aim of register allocation is to reduce the number of loads and stores to main memory. These are costly, especially when they cause a cache miss. The idea is to cache values in the processor's fast register file, so they do not need to be loaded

or stored from main memory as often. Register allocation is very similar to the graph-colouring problem, which is NP-complete.

There are several methods in the literature. The most well-known is the register colouring algorithm [**?**]. This generalises the problem to that of colouring a graph. Heuristics are used in an attempt to obtain the optimal colouring. The linear-scan algorithm [**?**] results in less optimal code compared to register colouring, but runs in linear time and hence is popular in modern JIT compilers, for example Mono.NET [**?**].

The Lua VM allows a maximum of `MAXVARS` local variables, set to 250 by default. The x86 architecture has 8 general-purpose machine registers, each capable of holding 4 bytes. Since Lua values can use up to 12 bytes, i.e. 3 x86 registers, there are only enough machine registers to allocate a single Lua value at a time. I decided that the complexity costs of implementing register allocation for any local variables would outweigh the benefits.

However, floating-point values may be stored on the x87 FPU stack (also only 8 slots). Register allocation would certainly improve the performance of numerically intensive code if it was performed only on floating-point values. Unfortunately, the x87 FPU makes implementing register colouring or linear-scan awkward because it requires that one operand of every operation is always at the top of the stack. Given the time available, it will suffice to implement a simple register allocation algorithm, which operates only within basic blocks and should give a reasonable allocation of registers.

Type Propagation Analysis

A program written in a dynamically-typed language like Lua is usually slower than one written in an equivalent statically-typed language. This is due in part to the additional overhead of continually checking and copying the type tags of values. In the JIT compiler, the handling code for each possible type may inflate the generate native machine code.

This problem does not exist in statically-compiled languages. Type errors are caught at compile-time instead of at run-time, hence there is no need to check types every time a value is used. On the *lua-l* mailing list, Reuben Thomas suggested a statically-typed version of Lua [7] for its performance benefits. Implementing this would be out of the project's scope but a similar idea involving the reduction of type checks could be used by the JIT compiler.

In most cases, the type of a variable remains the same for most of its lifetime. If we are certain that a variable has particular type during some program interval, then there is no need to check its type every time an operation is performed on it. Instead, the type can be checked during JIT compilation. There is a subtlety here: if a type error is found, consistency with the original interpreter should be maintained and the error should be thrown at the point where it occurred. Lua's metamethods mean that type errors may not cause the program to halt immediately but could instead trigger some desirable user-defined operation.

As well as eliminating some type checks, unnecessary copying of type tags could be reduced. If the source and destination registers are known to have the same type, then there is no need to copy the type tag. Lua's `Value` type takes up 12 bytes, which means 3 copying operations on a 32-bit architecture. However, the only type which uses all 12 bytes is the `NUMBER` type (8 bytes for the `double` and 4 bytes for

the type tag, see Figure 1.2). Thus if the type of a variable is not `NUMBER`, only a single copying operation is necessary.

Data-flow analysis may be used to find a conservative approximation to the possible types that a variable may have in some program interval. *Reaching definitions* [**?**] is a very similar problem; that of finding which definitions of variables reach various points in the program.

Code Generation

This module performs the final translation into executable native machine code, once all optimisation phases have been completed. The native machine code should observe the function calling conventions of the compiler in use, in this case GCC, which uses the `__stdcall` conventions [**?**].

2.3 Languages and Tools

2.3.1 Implementation Language

The existing Lua interpreter is written in portable ANSI C [**?**] for maximum portability between platforms. Due to the low-level nature of the JIT compiler, as well as for better integration with the existing VM, this was the most sensible language to use. Although I was only targeting the x86 architecture, I kept the code ANSI-compliant for better portability between x86 compilers.

My primary development platform was Linux, so I mainly used the GCC [**?**] C compiler during development. I also successfully tested other compilers such as *tcc* and *lcc*, and operating systems such as Windows XP and FreeBSD (see Section 4.2).

2.3.2 Further Tools

I used several UNIX utilities during development:

- *vim* to edit source code.
- *nasm* to assemble test programs written in assembly in order to examine the resulting native machine code.
- *gdb*, the GNU debugger, to debug the native code generated by the JIT compiler.
- *ChunkSpy*, a Lua disassembler, to view the instruction sequences generated by the Lua compiler.
- *CVS* to store a history of changes to the source code during development. This CVS repository was stored on the PWF as a backup (see Section 2.5).
- *Crontab*, an automated scheduler, to back up the CVS repository automatically.
- *LaTeX* would be used to typeset my dissertation so I familiarised myself with its syntax. I used the teTEX distribution on Linux.

```
-- use generator functions

function generatefib (n)
  return coroutine.wrap(function ()
    local a,b = 1, 1
    while a <= n do
      coroutine.yield(a)
      a, b = b, a+b
    end
  end)
end

for i in generatefib(1000) do print(i) end
```

Figure 2.4: Lua test program to calculate the Fibonacci sequence.

```
assert(not(1<1) and (1<2) and not(2<1))
assert(not('a'<'a') and ('a'<'b') and not('b'<'a'))
assert((1<=1) and (1<=2) and not(2<=1))
assert(('a'<='a') and ('a'<='b') and not('b'<='a'))
assert(not(1>1) and not(1>2) and (2>1))
assert(not('a'>'a') and not('a'>'b') and ('b'>'a'))
assert((1>=1) and not(1>=2) and (2>=1))
assert(('a'>='a') and not('a'>='b') and ('b'>='a'))
```

Figure 2.5: Extract from the regression test suite, for testing the comparison operators.

2.4 Acceptance Tests

The Lua source code is distributed with a collection of 20 simple test programs, which are meant to ensure Lua is working as well as to show what Lua programs look like. Figure 2.4 shows one of the test programs, which calculates numbers from the Fibonacci sequence which are less than 1000. These test programs do not constitute a comprehensive regression test suite but it would be infeasible to write my own in the time available. Roberto Ierusalimschy kindly provided me with a copy of the Lua development team's internal regression test suite. An extract from the test suite is shown in Figure 2.5, used for testing the comparison operators.

2.5 Risk Assessment

Data loss and failure to complete on-time are important risks associated with software engineering projects. In anticipation of this, contingency plans were made in case something went wrong.

The biggest worry was probably data loss. All dissertation work and source code files were stored in a CVS repository on the PWF. This served as a backup medium in case my personal machine failed and also as a way of allowing rollbacks in case of programming errors. The repository was updated regularly and backed up automatically to the Pelican fileserver and to data locations in the US every night, using an automated scheduler (crontab).

An Intel x86 machine and an ANSI C compiler is all I needed to work on the project so I could continue easily on a PWF computer in the event of a hardware failure on my personal computer.

The other concern was how achievable my project goals were in the time available. I was certain that a basic JIT compiler could have been implemented at the very least, even if it only supported a handful of Lua instructions. However, I was aware that unforeseen difficulties could have arisen due to subtleties in the workings of the Lua VM, and caused the implementation to take longer than expected. If this had happened then I would have discarded some of the additional optimisation modules that were originally planned. In the worst case, I would have only implemented the JIT compiler without any optimisations. This would have still resulted in some speedup due to the removal of the instruction decoding overhead.

Each module was implemented with testing in mind and I also made sure the code was ANSI-compliant for maximum portability between C compilers and operating systems on the Intel x86 architecture.

Chapter 3

Implementation

3.1 Introduction

A just-in-time (JIT) compiler was developed based on the standard Lua runtime environment, incorporating two main optimisations: type propagation and register allocation. An incremental development model was used; the first prototype of the JIT compiler system was completed very quickly, supporting a handful of Lua instructions and falling back to the standard interpretive VM for the others. This could be tested straight away on benchmarks and example Lua programs (see Section 4.2). Support for more instructions was added gradually, as envisaged in the project plan. Finally, register allocation and type analysis were implemented.

I updated a log book on a daily basis while performing the project work; this has been a great aid in writing up this dissertation.

3.1.1 System Design

The JIT compiler consists of a number of phases, as seen in Figure 2.2. The intermediate Lua bytecode is analysed by several stages, and finally generates executable native machine code using a code generator.

The simplest possible JIT compiler essentially takes a sequence of intermediate bytecode and translates it directly into executable native machine code. This requires a code generator module, which maps each Lua instruction into a corresponding sequence of native machine code.

Figure 3.1 has been modified from Figure 1.5 to show how the JIT compiler fits into the execution model of the Lua VM.

The standard Lua VM interpreter is invoked via the `luaV_execute()` function so this is the ideal place to plug in the JIT compiler and other optimising modules.

3.2 Initial Prototype

The first priority was to implement translation from Lua bytecode to executable native machine code. A simple code generator was completed, which supported the most straightforward Lua instructions to begin with — those that only manipulate data. Further optimisation phases were added later one by one.

The system was incrementally tested as a whole during development, using the system test cases described in Section 4.2.

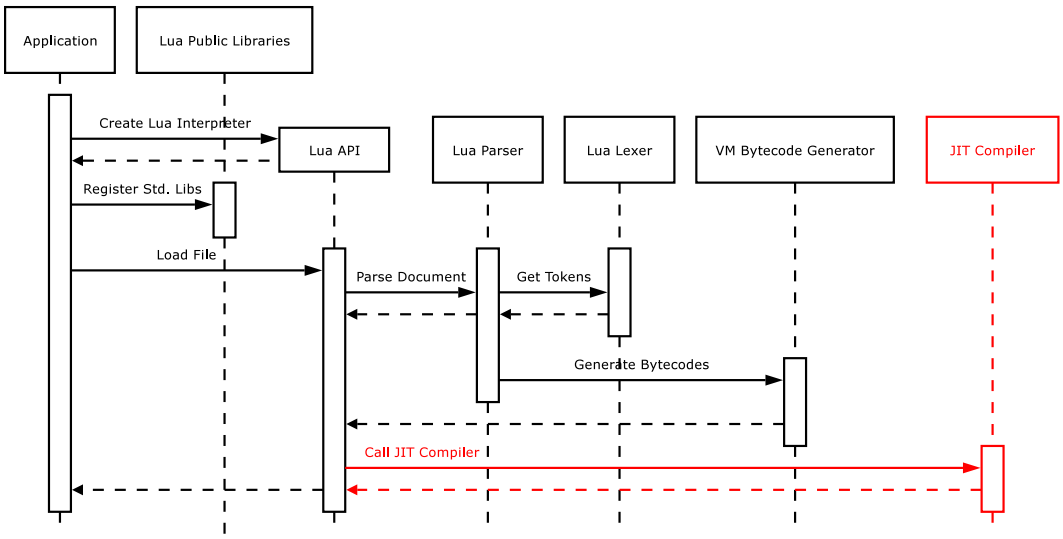

Figure 3.1: The Lua VM execution model, including the JIT compiler.

0 1 2 3 4 5	6 7 8 9 10 11 12 13	14 15 16 17 18 19 20 21 22	23 24 25 26 27 28 29 30 31
OP	A	B	C
OP	A	Bx	
OP	A	sBx	

Figure 3.2: Format of a Lua Instruction

3.3 Code Generation

For each Lua instruction a corresponding code generating function was written. A fallback mechanism was used so that the JIT compiler could be tested on arbitrary Lua programs, even if they used unimplemented instructions. Unimplemented instructions were handled by the standard interpretive VM.

Initially, development proceeded by disassembling test programs and elements of the standard Lua interpretive VM to help compose the corresponding native machine code for each Lua instruction. In the JIT compiler code, each Lua opcode had a corresponding C function, which filled a buffer with specialised machine code. The machine code was entered manually, and this quickly became tedious and error-prone. The code was not self-documenting and was hard to read and understand. Figure 3.3 shows an extract from the initial prototype, of the code generating function for the `MOV` instruction.

After some research into the programming practices of existing JIT compilers, I discovered that the Mono.NET project [**?**], an open source version of Microsoft's .NET development platform, has a useful set of C macros for generating x86 machine code, which I was able to re-use and modify for my project. The equivalent to Figure 3.3 using the macros is shown in Figure 3.4. An example C macro from Mono.NET is shown in Figure 3.5. These macros resulted in much more readable and maintainable code. They did not completely cover the entire x86 instruction set, so several extra instruction macros had to be added.

Many Lua instructions involve calling some core library functions, to manipulate strings and tables for instance. These library functions were not usually inlined by

```
unsigned char *emit_MOVE(Instruction i, unsigned char *fun)
{
    int regA = GETARG_A(i) * sizeof(TObject);
    int regB = GETARG_B(i) * sizeof(TObject);

    *fun++ = 0x8b;          /* mov eax,[ebx+regB]   */
    *fun++ = 0x83;
    *(int *) fun = regB;
    fun += 4;

    *fun++ = 0x8b;          /* mov ecx,[ebx+regB+4] */
    *fun++ = 0x8b;
    *(int *) fun = regB + 4;
    fun += 4;

    *fun++ = 0x89;          /* mov [ebx+regA],eax   */
    *fun++ = 0x83;
    *(int *) fun = regA;
    fun += 4;

    *fun++ = 0x8b;          /* mov edx,[ebx+regB+8] */
    *fun++ = 0x93;
    *(int *) fun = regB + 8;
    fun += 4;

    *fun++ = 0x89;          /* mov [ebx+regA+4],ecx */
    *fun++ = 0x8b;
    *(int *) fun = regA + 4;
    fun += 4;

    *fun++ = 0x89;          /* mov [ebx+regA+8],edx */
    *fun++ = 0x93;
    *(int *) fun = regA + 8;
    fun += 4;

    return fun;
}
```

Figure 3.3: The initial prototype used machine code directly in the code generator.

```
unsigned char *emit_MOVE(Instruction i, unsigned char *fun)
{
    StkId ra = (StkId) (GETARG_A(i) * sizeof(TObject));
    StkId rb = (StkId) (GETARG_B(i) * sizeof(TObject));

    x86_mov_reg_membase(fun, X86_EAX, X86_EBX,          &rb->tt,          4);
    x86_mov_reg_membase(fun, X86_ECX, X86_EBX,          &rb->value,       4);
    x86_mov_membase_reg(fun, X86_EBX, &ra->tt,          X86_EAX,          4);
    x86_mov_reg_membase(fun, X86_EDX, X86_EBX,          &rb->value.b + 1, 4);
    x86_mov_membase_reg(fun, X86_EBX, &ra->value,       X86_ECX,          4);
    x86_mov_membase_reg(fun, X86_EBX, &ra->value.b + 1, X86_EDX,          4);

    return fun;
}
```

Figure 3.4: The code generator using macros.

```
#define x86_mov_reg_membase(inst,reg,basereg,disp,size) \
    do {    \
        switch ((size)) {   \
        case 1: *(inst)++ = (unsigned char)0x8a; break; \
        case 2: *(inst)++ = (unsigned char)0x66; /* fall through */ \
        case 4: *(inst)++ = (unsigned char)0x8b; break; \
        default: assert (0);    \
        }   \
        x86_membase_emit ((inst), (reg), (basereg), (disp));    \
    } while (0)
```

Figure 3.5: An example C macro from Mono.NET's `x86-codegen.h`.

the JIT compiler since it would unnecessarily inflate the compiled native machine code as well as being difficult to maintain.

For each Lua instruction, the corresponding implementation in the interpreter was carefully examined and essentially translated into assembly language by hand. There are so few x86 general-purpose registers that it was not felt to be worthwhile to perform automatic register allocation to cache Lua variables in registers. This would be different for architectures with more registers. I did perform register allocation for floating point variables on the stack, however.

3.3.1 Instruction Scheduling

Modern CPU architectures have long instruction pipelines. In order to improve performance on these machines, it was necessary to improve instruction-level parallelism by rearranging the order of instructions. They were rearranged so that a value is not used straight after it is computed, which may cause a pipeline stall. Care was taken so that the original meaning of the code was preserved. Intel's *Optimisation Reference Manual* [**?**] describes code optimisation techniques that can be used to finely tune applications for Intel processors.

Instruction scheduling was effectively done by hand in the code-generating functions for each Lua instruction. No inter-operation scheduling was performed, however, and this is a possible extension to the project. Since modern superscalar processors reorder instructions on-the-fly and perform dynamic instruction scheduling, this optimisation was not likely to make a lot of difference so no more time than necessary was spent on it.

3.3.2 Data Instructions

The simplest Lua instructions are `MOV` and `LOADK`, which transfer data between local variables and load constant values into local variables respectively. Essentially they copy a Lua `TObject` value from one memory location to another.

The number of copying operations can be reduced using the following observations:

1. For the `LOADK` instruction, we know that the constant table is invariant, thus the constant value can be injected directly into the emitted native machine code. This reduces the number of accesses to memory thus reducing latency at the expense of slightly increased code size.

2. If the types of the source and destination registers are known to be the same (using type analysis, see Section 3.6), the type field does not need to be copied.

3. Since a Lua value is a union type, the value field doesn't always take up 2 words. Again, type analysis is useful here because only one word needs to be copied for the value if the value doesn't have a numeric type.

3.3.3 Arithmetic Instructions

Lua has a single numerical data type, `number`. The meaning of `number` can vary according to a compile-time option, which is typically set to use double-precision floating point numbers.

From a performance point of view, this seems strange at first, since floating-point operations are typically thought to be slower than integer calculations, and lack precision for large values when compared with equivalent sizes. However, the Lua community argues that modern floating-point implementations are just as fast as their integer counterparts, and point out that IEEE 754 64-bit double-precision floating-point numbers can also represent a greater range of integers exactly than a 32-bit integer (-2^{53} to $+2^{53}$ compared with -2^{32} to $2^{32} - 1$, according to the community Wiki [**?**]).

In the interest of making numerical calculations much faster, I considered adding a new integer data type to the Lua core. However, this would be too costly to implement and the performance gains would be small, especially taking into account Amdahl's law since typical Lua programs are unlikely to be highly numerically intensive.

When implementing the floating-point instructions, I had a choice of possible instruction set extensions: the older x87 FPU is supported by practically all Intel x86 processors, 386 and above; but the downside is its stack-based model, which makes programming it harder. The alternative is to use SSE[1], which have a flat set of registers instead of a stack. However, SSE2 is required for double-precision floating-point arithmetic; for Intel this extension is only supported on the Pentium 4. As for speed, SSE2 double-precision operations are no faster than the x87 FPU on the Pentium 4 (see timings in Agner Fog's *Pentium Optimisation Manual* [**?**]. After weighing up these options I decided to use the x87 FPU because of its backwards-compatibility and double-precision support. SSE2 support was left as a possible future extension, as was exploitation of the SIMD aspect of these instructions.

It should be noted that the x87 FPU actually uses 80-bit extended-precision internally and so will give slightly different results from using the standard Lua VM, since this extra precision is lost when flushing floating-point values to memory. The standard Lua VM flushes values immediately after every operation, converting them back to 64-bit precision. The register allocator in the JIT compiler may cause values to remain in 80-bit precision across multiple operations before flushing to memory, causing some iterative numerical algorithms to behave differently.

Simple Strength Reduction

Expensive operations often have special cases which can be implemented using cheaper instructions depending on the target instruction set. Strength reduction [**?**] refers to the replacement of these expensive operations by equivalent but cheaper operations. For example, it may be cheaper to compute $v + v$ instead of $v * 2$ on an architecture where addition is cheaper than multiplication. In some cases, the new operation might cost the same in terms of time but have a shorter machine code size. This is also advantageous since we are interested in keeping the memory usage down as much as possible. Shorter code length often leads to performance improvements e.g. because it is more likely to fit in the cache and it is easier for the CPU to decode.

In the case of Lua, `FLD1`, `FLDZ`, `FLDPI` are inserted where possible to avoid the expense of loading constants from memory. It was deemed unnecessary to implement strength reductions like $1^x \to 1$ or $x^1 \to x$ in the JIT compiler since these could have been done by Lua's static compiler.

[1] SSE stands for Streaming SIMD (Single Instruction Multiple Data) Extensions

Register	Description
EAX	Accumulator
EBX	Pointer to base of local variable register file
ECX	Loop counter
EDX	General-purpose
EDI	General-purpose
ESI	General-purpose
EBP	Pointer to base of stack
ESP	Stack pointer

Table 3.1: General-purpose registers available on Intel x86.

3.3.4 Function Calls

These were implemented last of all, and turned out to be the most difficult. One difficulty is that the C stack needs to be kept constant in order to preserve coroutine semantics, since `coroutine.yield()` assumes that the C stack is constant. Every time a function call is encountered, the native machine code returns control to the VM. The VM then calls the JIT compiler module again or uses its interpretive mode to execute the called function, which may again call the JIT compiler during its execution.

3.4 Register Allocation

The simple register allocator works on basic blocks of arithmetic instructions by simulating an x87 FPU stack. When the code generator needs a floating-point value loaded, the register allocator checks the state of the simulated stack. If the stack is full then it pops the stack and, if the value is marked *dirty*, it emits the native machine code for flushing a value to its memory location. When a floating-point value is stored, it is pushed onto the stack, once space has been made, and marked *dirty* so that it will be flushed to memory eventually.

The simulated stack is completely flushed in the following cases:

- At the end of a basic block, to preserve the consistency of the simulated stack.
- If a Lua instruction is encountered that calls a C library function, requiring the physical x87 FPU stack to be empty in compliance with calling conventions.

Table 3.1 summarises the general-purpose registers available on the x86 architecture and their usage in the JIT-compiled block.

3.4.1 Control Flow Instructions

In order to implement the control flow instructions a flow graph is constructed first to make tracking of jump targets easier.

These were slightly more complicated as they involve tracking jump/branch targets throughout the code. It gets slightly tricky when you have forward jump target because an unsupported instruction might appear in between and so you have to flush.

3.5 Basic Blocks and Flow Graphs

Basic blocks were represented by a structure consisting of a count of the number of instructions in the block, followed by a pointer to the leader (first instruction) of the block, and by the lists of predecessors and successors of the block.

Given a flow graph $G = \langle N, E \rangle$, with node set N and edge set $E \subseteq N \times N$, we define predecessor and successor sets as follows:

$$\begin{aligned} Succ(b) &= \{n \in N \mid \exists (b, n) \in E\} \\ Pred(b) &= \{n \in N \mid \exists (n, b) \in E\} \end{aligned}$$

1. Firstly determine the set of *leaders*, the first statements of basic blocks. The following rules are used:
   (a) The first statement is a leader.
   (b) Any statement that is the target of a conditional or unconditional jump is a leader.
   (c) Any statement that immediately follows a jump or conditional jump statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

3.6 Type Analysis

Types of registers in Lua's virtual machine can be conservatively approximated using data-flow analysis. This problem is similar to *reaching definitions* [**?**], which involves finding which definitions of variables reach various points in the program.

The reaching definitions problem can be expressed as the following data-flow equations:

$$\begin{aligned} REACHout(i) &= (REACHin(i) \cap PRSV(i)) \cup GEN(i) \\ REACHin(i) &= \bigcup_{j \in Pred(i)} REACHout(j) \end{aligned}$$

Here, $REACHin(i)$ and $REACHout(i)$ represent the variable definitions that reach the beginning and end of an instruction node i, respectively. We define sets $GEN(i)$ that give the definitions generated by node i, and sets $PRSV(i)$ that give the definitions preserved by node i.

Analogously, we let $TYPESin_r(i)$ and $TYPESout_r(i)$ represent the set of possible types of a virtual machine register r before and after an instruction node i, respectively. The following data-flow equations may be used to infer the possible types of the virtual machine registers.

$$\begin{aligned} TYPESout_r(i) &= (TYPESin_r(i) \cap PRSV_r(i)) \cup GEN_r(i) \\ TYPESin_r(i) &= \bigcup_{j \in Pred(i)} TYPESout_r(j) \end{aligned}$$

The sets $GEN_r(i)$ give the possible types *generated* by block i, for a virtual machine register r. The sets $PRSV_r(i)$ represent the possible types of variables *preserved* by block i, for a virtual machine register r.

Recall that Lua only defines 8 types. The sets of possible types are implemented using vectors of 8 bits, with each bit position representing a type. Thus for all virtual machine registers, their possible types are represented by a byte array. The union operation then becomes a simple bitwise disjunction, and intersection a bitwise conjunction:

$$\begin{aligned} TYPESout_r(i) &= GEN_r(i) \vee (TYPESin_r(i) \wedge PRSV_r(i)) \\ TYPESin_r(i) &= \bigvee_{j \in Pred(i)} TYPESout_r(j) \end{aligned}$$

The type analysis is performed only on the virtual machine's local register file, i.e. local variables. Global variables pollute the space of possible types since their types may be changed during Lua function calls or C API functions. Some inter-procedural analysis would need to be done for global variable types to be inferred properly.

Iterative analysis is used to perform type analysis as this is the easiest method to implement. The following is pseudocode for an iterative solution to the data-flow equations:

do
  $change \leftarrow$ **false**
  for each block B **do**
    $in[B] \leftarrow \bigcup_{P \in Pred(B)} out[P]$
    $oldout \leftarrow out[B]$
    $out[B] \leftarrow gen[B] \cup (in[B] - kill[B])$
    if $out[B] \neq oldout$ **then** $change \leftarrow$ **true**
while $change$

Intuitively, types are propagated as far as possible without them being killed. The algorithm is guaranteed to terminate eventually because $out[B]$ never decreases in size for any B. This is because the rules for the iteration are monotone, i.e. they never change a 1 to a 0 in $out[B]$.

In the current implementation, type information about values only comes from loading constants and performing arithmetic operations. The analysis would be even more effective if the types of function arguments were taken into account. This is discussed as an extension in Section 5.1.3.

See Appendix A for an extract of C code that performs type analysis.

3.6.1 Constants

When constants are loaded from Lua's constant table (each closure has its own) with `LOADK`, the JIT compiler injects the constant value directly into the native instruction sequence. This should yield better performance than loading the constant value from the constant table in memory at run-time, but will increase the native code size slightly. The types of constants are always known so again 12 bytes only need to be copied if the type is `NUMBER`.

3.7 Caching Compiled Code

The performance improvement would not have been possible without a suitable method for caching the compiled native machine code to be reused in later iterations of the same Lua code. The hash table data structure immediately sprung to mind due to its amortised time complexity of $O(1)$ for both insertions and lookups. However, hash tables usually have a high constant multiplier hidden in the time complexity.

A hash table was used in the initial prototype, but it was discovered that a faster method could be used. The JIT compiler module is called at the entry and exit points of functions. Using this fact, the compiled native machine code for each function was cached in Lua's existing function descriptor data structures, so that the existence of JIT-compiled code could be checked very quickly.

3.8 Debugging

The GNU debugger `gdb` was very useful for this because it can disassemble arbitrary sequences of code in memory and set breakpoints on them. A typical debug session would involve single-stepping through each instruction and viewing the state of the processor register file.

Another useful technique was to output a flow graph in the DOT graph description language to be converted into a graphical form by the `dot` tool. For example, the following short snippet of Lua code would result in the DOT graph in Figure 3.6, and the graphical form in Figure 3.7:

```
for i = 1, 10 do
    print(i)
end
```

The resulting flow graph could then be examined and checked manually on paper.

I found *Drawing Graphs with Dot* [**?**] a useful reference when learning the DOT language.

3.9 JIT Heuristics

As discussed in Section 2.2.2, it is generally not worthwhile to JIT-compile regions of code that are only executed once. The compilation overhead is usually greater than the performance gained from executing the compiled native code.

A simple heuristic was implemented: only call the JIT compiler the second time a region of code is executed. However, the benchmarks used for testing the JIT compiler spent over 99% of their time repeatedly executing in the main loop of the benchmark. So much so, in fact, that the compilation overhead associated with JIT-compiling code only executed once became negligible. The fine-tuning envisaged in the original plan was unnecessary.

Another issue worth considering was what constituted a region to be JIT-compiled. At first, when only the simplest Lua instructions had been implemented, the JIT compiler would compile as much as it supported. Later on in the development cycle, when support for control-flow instructions and function calls was

```
digraph G {
    graph [labeljust = l  ];
    node  [shape     = box];
    "8079d20" [label = "LOADK   R[0]:x K[0]:8\l" +
                       "LOADK   R[1]:x K[0]:8\l" +
                       "LOADK   R[2]:x K[0]:8\l" +
                       "FORPREP R[0]:8 R[3]:x\l"];
    "8079d20" -> "8079ae0";
    "8079b40" [label = "GETGLOBAL R[4]:0 G[0]:0\l" +
                       "MOVE      R[5]:x R[3]:8\l" +
                       "CALL      R[4]:0  [2]:0 [1]:0\l"];
    "8079b40" -> "8079ae0";
    "8079ae0" [label = "FORLOOP R[0,1,2]:8,8,8 \l"];
    "8079ae0" -> "8079b40";
    "8079ae0" -> "8079bb0";
    "8079bb0" [label = "RETURN\l"];
}
```

Figure 3.6: Example flow graph in DOT graph format

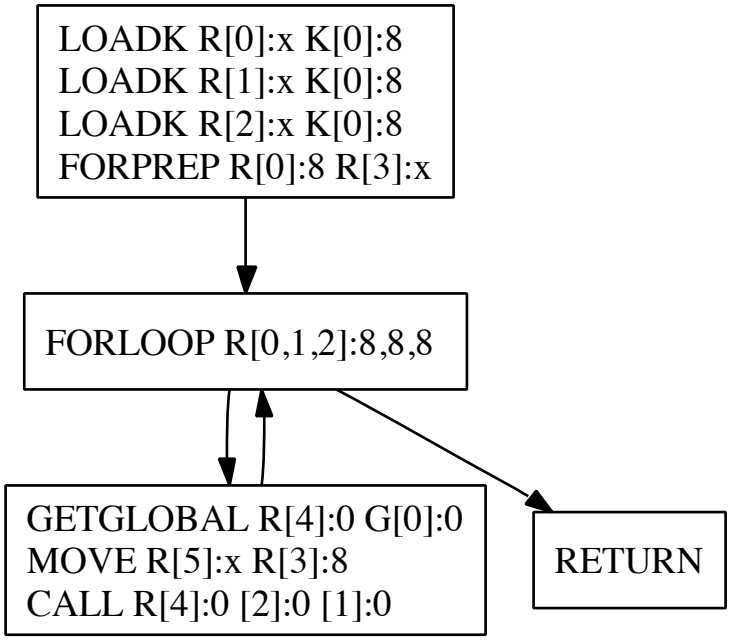

Figure 3.7: Example flow graph graphics generated by `dot`

added, entire functions were compiled if all their instructions were supported. Theoretically it might be unwise to compile a very large function because of the high memory usage that would result. This is unlikely to arise in practice though, and all benchmarks used were relatively short (less than 1000 lines).

More advanced heuristics would analyse the performance statistics of the code at runtime and try to estimate how much the JIT compiler saves on each iteration. For instance, we might give higher priority to compiling loops because they are more likely to be executed repeatedly.

3.10 Testing

A simple test harness was written to compare the results of running the test programs on the original interpreter with the optimised one. The test harness was simple to write using Lua and took of a list of test programs to run, then ran the old and new interpreters in turn, saving their outputs and then comparing them.

Roberto Ierusalimschy of the Lua development team kindly allowed me to use their internal regression test suite to test the correctness of the VM. Their test suite consisted of many short unit test cases written in Lua. These were designed to catch implementation bugs and would have taken a long time to write independently.

3.11 Benchmarking

Benchmarks can often be misleading; tremendous performance increases in special cases do not necessarily imply great performance increases in typical real-world programs.

There is a well-known set of benchmarks known as the Great Computer Language Shootout [**?**, **?**]. In order to compare performance before and after addition of the JIT compiler, I used a selection of these benchmarks. Some of them were inappropriate and tested areas such as I/O performance, which wouldn't be affected by this JIT compiler. These pre-existing benchmarks saved a substantial amount of work and also allowed easy comparison with other languages.

The LINPACK benchmark [**?**] is widely used for measuring floating-point performance on different systems. Based on an implementation in Java, I wrote a version for Lua, included in Appendix B. The conversion was very straightforward, and as expected the benchmark showed a reasonable performance increase when the JIT compiler was enabled, as we will see in Section 4.3.

Chapter 4

Evaluation

The JIT compiler was implemented as a portable module for the Lua core, and can be easily enabled or disabled if required. All test cases were completed successfully, showing that the operation is consistent with the original, unmodified interpreter. The modular design of the JIT compiler itself allows the various optimisation phases to be turned off if required and also allows finer-grained control of the JIT when embedded in other programs.

During the implementation phase, the system was tested incrementally as a whole while it was being developed. For example, when implementing support for Lua's arithmetic instructions, small test programs were written in Lua to ensure arithmetic was done correctly. The test programs were executed both using the old, unmodified interpreter and the JIT compiler, and the results compared. The majority of bugs were caught in this way early on in the implementation phase.

4.1 Module Testing

Testing modules individually was difficult because of interdependencies; this is often the case with compilers. It was easier to test the system incrementally as a whole and catch bugs that way. Some of the optimisation modules could also be disabled to help narrow down the location of a bug.

Flow Graph Generation

Although this module was fairly simple, it was important that it worked correctly since subsequent data-flow analysis phases depended on it. Flow graph generation was tested using a number of test cases designed to exercise as many interesting control flows as possible. These were then checked by hand; the debugging modules helped with this by translating the flow graph into the DOT graph description language, which could be used to generate a graphical version of the flow graph.

Type Propagation

Similarly, this module was tested using a number of simple test cases and checked by hand.

Program	Description
`life.lua`	The game of Life by John Conway.
`bisect.lua`	Bisection method for solving non-linear equations.
`cf.lua`	Temperature conversion table (Celsius to Fahrenheit).
`echo.lua`	Echo command line arguments.
`env.lua`	Read environment variables as if they were global variables.
`factorial.lua`	Compute factorials.
`fibfor.lua`	Fibonacci using generator functions.
`fib.lua`	Fibonacci function with cache.
`hello.lua`	Traditional Hello World program.
`printf.lua`	An implementation of printf.
`readonly.lua`	Make global variables readonly.
`sieve.lua`	The sieve of of Eratosthenes programmed with coroutines.
`sort.lua`	Two implementations of a sort function.

Table 4.1: Test Programs

4.2 System Testing

As well as the small programs used during incremental testing, more substantial programs were used to put the system through its paces, listed in Table 4.1. The results of executing the programs using the JIT compiler were automatically compared with those produced with the original Lua interpreter. A shell script executed the old and new interpreters in turn, saving their outputs and then comparing them. This was called from the makefile every time the source code was compiled so that bugs could be caught as early as possible.

All test cases worked perfectly with exactly the same results as the original interpreter.

In addition, more stringent testing was done using the Lua development team's internal regression test suite. These tests are used to help ensure the correctness of the standard Lua VM. They cover all the operands as well as the standard library functions. The JIT compiler passed all of these tests with flying colours.

The modified interpreter was also tested on a variety of x86 operating systems. As long as an ANSI-compliant C compiler is used, and it follows the `__stdcall` calling conventions, the JIT compiler should work on any x86 operating system. It was tested using the system test cases above and was found to work perfectly on Microsoft Windows XP, FreeBSD and NetBSD as well as Linux.

4.3 Benchmarks

The principle objective of this project was to improve the runtime performance of Lua. To show that this had been achieved, I used various benchmark programs and collected timing and memory usage data for the standard Lua interpreter and the JIT compiler. These were mainly benchmarks from The Great Computer Language Shootout (see Section 3.11).

Benchmark	Standard	JIT (non-optimising)			JIT (optimising)		
	Time (s)	Time (s)	Speedup	Code size	Time (s)	Speedup	Code size
arith-loop	17.09	4.66	3.67	16939	1.63	10.48	3202
mandelbrot	15.75	3.23	4.88	3018	2.60	6.06	2687
nested-loop	9.67	2.33	4.15	1855	2.39	4.05	1849
linpack	12.87	8.12	1.58	49142	6.93	1.86	48838
matrix	7.15	5.25	1.36	5523	5.16	1.39	5900
nbody	4.92	3.68	1.34	14009	3.61	1.36	16787
life	6.02	4.80	1.25	17228	4.79	1.26	17281
ackermann	3.20	2.80	1.14	2685	2.80	1.14	2685

Table 4.2: Benchmark timings

n	Standard (s)	JIT (unoptimised)	JIT+regalloc	JIT+regalloc+typeprop
256	0.006	0.004	0.004	0.004
1024	0.015	0.007	0.007	0.006
4096	0.048	0.017	0.016	0.011
16384	0.177	0.058	0.052	0.037
65536	0.690	0.210	0.192	0.127
262144	2.750	0.822	0.745	0.488

Table 4.3: Mandelbrot benchmark timings for different n.

4.3.1 Results

Table 4.2 compares the time taken to run various benchmarks for the standard Lua VM, the non-optimising JIT compiler (i.e. register allocation and type propagation turned off) and the optimising JIT compiler. It also shows the size of the JIT compiler's native machine code buffer (in bytes), giving an indication of memory overheads. The timings are shown as a graph in Figure 4.1.

The mandelbrot benchmark was tested with different optimisation phases turned off and for different output resolutions with n being the total number of pixels. The timings are shown in Table 4.3, and the graph in Figure 4.2.

4.3.2 Analysis

Benchmarks can sometimes be misleading but in this case I believe they provide a valuable insight into the performance gains achieved by using a JIT compiler over an interpretive VM.

In the case of the *arith-loop* benchmark, we see a tremendous performance increase of over a factor of 10. However, this is an isolated special case and we should examine this result carefully to understand why this is so. The *arith-loop* benchmark consists of a single loop, containing a large number of arithmetic operations, repeatedly executed many times. This benefits greatly from register allocation in particular due to the intensive use of floating-point arithmetic. The type propagation analysis will also help improve performance by reducing the number of type tag checks and copies in arithmetic operations.

Other benchmarks do not show such a tremendous increase in performance, although the mandelbrot benchmark still shows significant gains. The lowest increase came from the ackermann benchmark, which is not surprising since no intra-

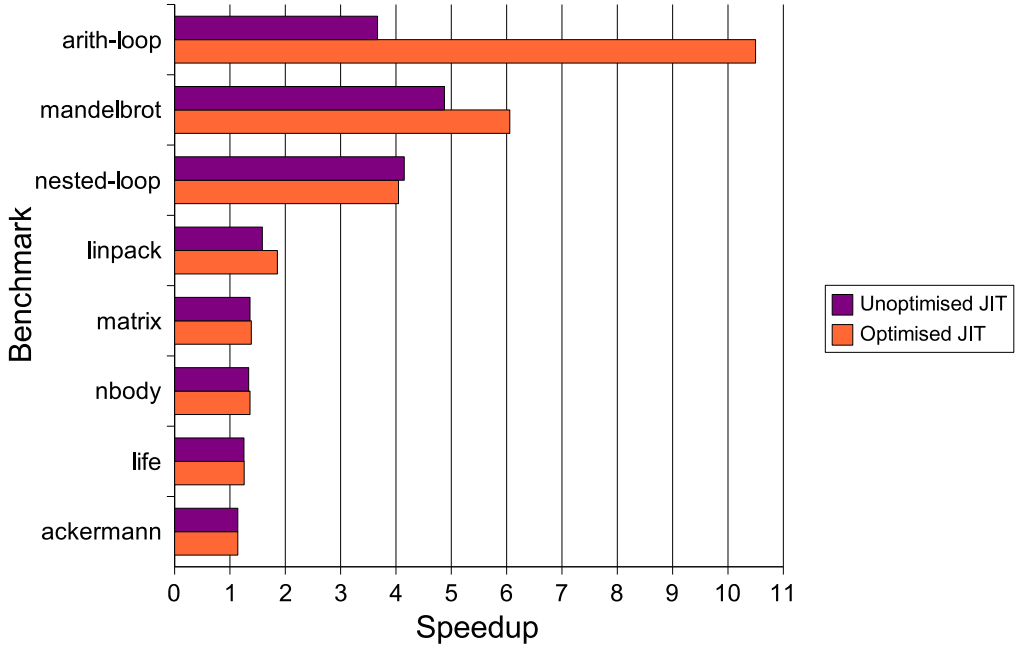

Figure 4.1: Benchmarks

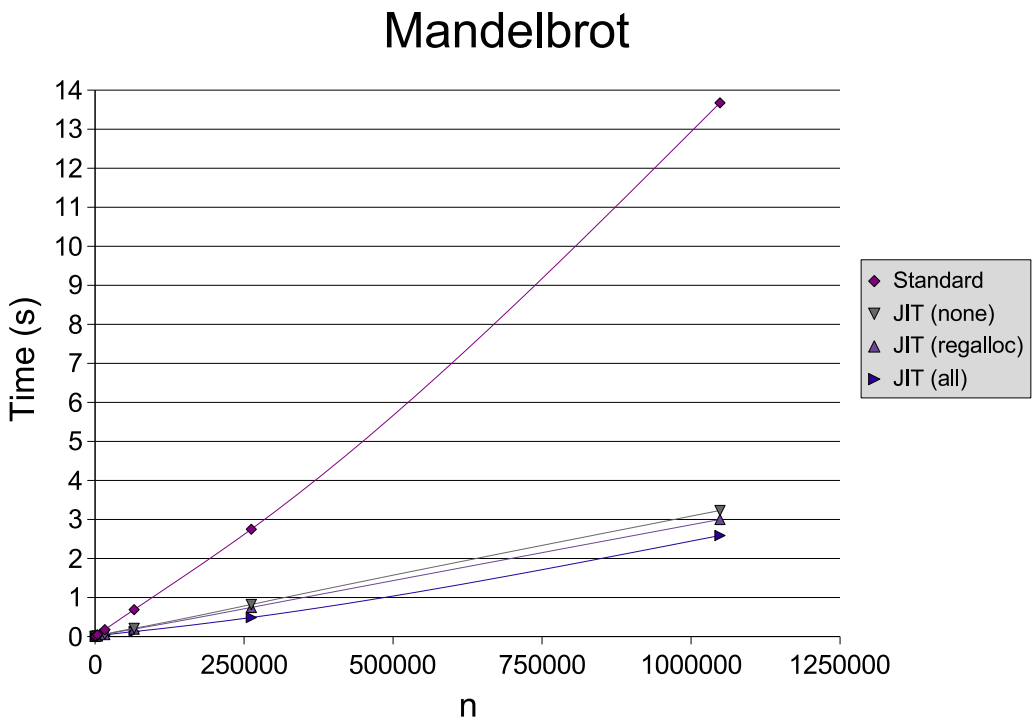

Figure 4.2: Mandelbrot Benchmark.

procedural optimisation was done and ackermann is a test of recursion performance.

Compared to the compact Lua bytecode, the native code buffer sizes of the JIT compiler are fairly large. This might be important in an embedded systems environment with tight memory constraints.

Overall, the benchmarks demonstrate that the original success criterion of performance increase has been met.

The effect of register allocation and type analysis was investigated more closely for the mandelbrot benchmark (see Figure 4.2). The additional performance gains are not as significant as those obtained using the pure JIT compiler over the interpretive VM, but this may be due to the benchmark spending most of its time performing numerical calculations.

4.4 Requirements Met

The original requirements laid out in Section 2.1 can be used to justify the success of the implementation. Here it is shown how each requirement has been satisfied.

1. The JIT compiler is completely consistent with the original interpreter, as shown by the regression tests.

2. Type propagation analysis for local variables was implemented and tested successfully. This enabled many type checks to be eliminated.

3. Simple heuristics for the JIT compiler were implemented.

4. The project has seen impressive performance improvements as demonstrated by the benchmarks.

5. The platform independent portions of the JIT compiler system was implemented in standards-compliant ANSI C. This was checked using the `-ansi` flag with `gcc`.

6. The Intel x86 architecture was targeted and the system was tested on several x86 operating systems: Linux, Windows XP, FreeBSD and NetBSD.

7. A modular design approach was employed, making it easier to test individual units as well as improving maintainability.

8. All code was clearly commented and formatted for greater readability and maintainability.

Chapter 5

Conclusions

I have successfully completed an optimising just-in-time (JIT) compiler for the Lua programming language, which resulted in considerable performance improvements as shown by benchmarks, and meets all the requirements initially laid out.

It was very satisfying to see the performance increase as a result of my work putting compiler optimisation theory into practice. Implementing the JIT compiler itself turned out to be more straightforward than I initially expected, but I also discovered how error-prone low-level programming is when using C and assembly language, and the difficulty of debugging machine code.

At first I was worried that Lua's source code might take too long to dissect and understand, but despite its extensive use of C macros for portability, Lua turned out to be well-implemented in a modular fashion. Thus it was very easy to augment the virtual machine with the JIT compiler modules. Lua's meticulous attention to portability also meant that there were no problems compiling it on various x86 operating systems.

The trouble with optimisation projects is that you can spend forever tweaking to squeeze more performance out of the system. The modular design of my JIT compiler system should allow further optimisation passes to be added easily if desired.

Just-In-Time Compilation of Dynamically-Typed Languages

Although JIT compilers have already been written for other dynamically typed languages, such as Python and Perl, my work reinforces the idea that JIT compilation is a worthwhile method for improving run-time performance of such languages. This is important if these languages are to be used in real-time applications that require high performance.

As discussed in the introduction, JIT compilation suffers from a number of drawbacks, in particular high memory usage and low portability. Higher memory usage is mainly a concern in embedded systems, and I believe that the speed increase makes up for these disadvantages when high runtime speeds are required.

Hindsight

In retrospect, the project went very well in terms of keeping to the timetable and plan. The original success criteria were attainable and the timetable gave sufficient time for thorough research to be undertaken at the beginning. Lua has a lot of

reference material available in the form of technical manuals, research papers and a book, all accessible on the Web. These were invaluable to me when getting acquainted with the language and the virtual machine's internals.

With hindsight one can usually see where time could have been better spent. In my case, much of the implementation time was spent on implementing code generation for as many Lua instructions as possible. Looking back, this time might have been better spent working on further optimisation phases, as they are more interesting from an academic point of view. On the other hand, it was necessary to implement most of the opcodes for the JIT compiler to work well on the general-purpose benchmark programs and produce a reasonable performance increase.

The modular design used throughout makes it easy for further optimisation phases to be added if necessary. Porting it to other machine architectures, however, would be a harder task since the code generator was x86-specific. Ideally, a target-independent code generator would be used such as vcode [**?**] but portability between machine architectures was not deemed important enough to pursue it further.

5.1 Future Work

A couple of possible extensions were mentioned in my project proposal, included in Appendix C, but at the time I didn't fully understand the intricacies of the problem at hand. The first idea was to implement a typed version of Lua. This would require fairly substantial modifications to the Lua core and was outside the scope of my project due to the time available. The other possible extension was to allow memory usage to be adjusted using a tuning parameter, causing the JIT compiler to fall back to the interpreter for more complex instructions. Unfortunately this also had to be abandoned because of time constraints.

5.1.1 Register Allocation

As discussed in Section 3.4, I used a simple register allocation algorithm for floating-point variables. This could be improved upon by replacing it with either register colouring, which gives more optimal results, or linear-scan [**?**], which is faster and is used in modern JIT compilers.

On a machine architecture with more general-purpose registers, such as the x86-64 which has 128, register allocation should be performed for other types of local variables as well as floating-point types.

5.1.2 Integer Arithmetic

Another optimisation to be considered is the use of integer arithmetic instead of floating point in some areas of code. This would allow arrays in Lua to be even faster because the values would no longer have to be checked to see if they are integers. Integers could be detected using abstract interpretation [**?**]. Tables could also be specialised for particular types in order to reduce type checks, because typically a table will only hold values of a single type.

5.1.3 Type Analysis

The type analysis module would be more effective if types of function arguments were taken into account. In the current implementation, type information about values only comes from loading constants and performing arithmetic operations. Depending on the types of its arguments, several versions of a function may be JIT-compiled and specialised for those types.

5.1.4 Traditional Compiler Optimisations

There are a number of remaining traditional compiler optimisations which could be implemented. These are all fairly cheap to execute and would not impact the performance of the JIT compiler greatly.

- Loop unrolling
- Prefetching
- Loop hoisting
- Constant propagation
- Common Subexpression Elimination

5.1.5 Other Architectures

In targeting the Intel x86 architecture much of Lua's portability was sacrificed. It would be interesting to implement code generators for other architectures too. Additional low-level intermediate code would be generated before the final translation into native code for a particular target architecture. This would also allow cheap, low-level peephole optimisations to be performed on the low-level code. Some work has already been done on retargetable JIT compilers, for example Jello [**?**], which is effectively a JIT compiler compiler.

5.1.6 Background Compilation

Various techniques exist for reducing the overhead of dynamic compilation. Krintz et. al. [**?**] evaluate the effectiveness of *lazy compilation* and profile-driven *background compilation* as a way of overlapping compilation with execution.

5.1.7 Ahead-of-Time Compilation

Just-in-time compilers are generally unsuited for embedded systems. If high performance is needed, then Lua code could be compiled *ahead-of-time* into a native machine code executable binary. The subsystems of the JIT compiler could be reused to generate the executable code. The advantage of *ahead-of-time* compilation is that it eliminates the dynamic compilation overhead by pre-compiling the code.

Appendix A

Code for Type Analysis

Included here for reference is the C code for performing type analysis on a Lua program flow graph.

```
void type_analysis(BasicBlock *entry, StkId k)
{
    const Instruction *pc;
    int c, d, changes;
    int ra, rb, rc;
    unsigned int *old_types;
    unsigned int *types
        = (unsigned int *) malloc(MAXVARS * sizeof(unsigned int));
    BasicBlock *bb;
    /* Initialise bit-vectors */
    for (bb = entry; bb != NULL; bb = bb->next_bb) {
        if (bb->t == NULL) {
            bb->t = (TypeState *) malloc(bb->code_len * sizeof(TypeState));
            for (c = 0; c < bb->code_len; c++) {
                bb->t[c].a = 0;
                bb->t[c].b = 0;
                bb->t[c].c = 0;
            }
            bb->types
                = (unsigned int *) malloc(MAXVARS * sizeof(unsigned int));
            for (c = 0; c < MAXVARS; c++) {
                /* register file may contain any types at the entry node */
                bb->types[c] = (bb == entry) ? ~0 : 0;
            }
        }
    }
    do {
        changes = 0;
        for (bb = entry; bb != NULL; bb = bb->next_bb) {
            old_types = bb->types;
            pc = bb->code;
            /* clear */
            for (c = 0; c < MAXVARS; c++) {
                types[c] = (bb == entry) ? ~0 : 0;
            }
            /* union with predecessors */
            for (c = 0; c < bb->in_count; c++) {
                BasicBlock *in_bb = bb->in_bb[c];
                for (d = 0; d < MAXVARS; d++) {
                    types[d] |= in_bb->types[d];
                }
            }
            for (c = 0; c < bb->code_len; c++) {
                const Instruction i = *pc++;
                ra = GETARG_A(i);
                /* Fill bit-vectors accordingly */
                switch (GET_OPCODE(i)) {
                    case OP_ADD:
                    case OP_SUB:
```

```
case OP_MUL:
case OP_DIV:
case OP_POW:
    rc = GETARG_C(i);
    bb->t[c].c = ISK(rc) ? bitmask(k[INDEXK(rc)].tt) : types[rc];
case OP_UNM:
    rb = GETARG_B(i);
    bb->t[c].b = ISK(rb) ? bitmask(k[INDEXK(rb)].tt) : types[rb];
    bb->t[c].a = types[ra];
    types[ra] = bitmask(LUA_TNUMBER);
    break;
case OP_MOVE:
    rb = GETARG_B(i);
    bb->t[c].b = types[rb];
    bb->t[c].a = types[ra];
    types[ra] = bb->t[c].b;
    break;
case OP_LOADK:
    bb->t[c].b = bitmask(k[GETARG_Bx(i)].tt);
    bb->t[c].a = types[ra];
    types[ra] = bb->t[c].b;
    break;
case OP_NOT:
    rb = GETARG_B(i);
    bb->t[c].b = ISK(rb) ? bitmask(k[INDEXK(rb)].tt) : types[rb];
case OP_LOADBOOL:
    bb->t[c].a = types[ra];
    types[ra] = bitmask(LUA_TBOOLEAN);
    break;
case OP_LOADNIL:
    rb = GETARG_B(i);
    while (ra <= rb) {
        types[ra] = bitmask(LUA_TNIL);
        ra++;
    }
    break;
case OP_EQ:
case OP_LT:
case OP_LE:
case OP_TEST:
    rb = GETARG_B(i);
    rc = GETARG_C(i);
    bb->t[c].b = ISK(rb) ? bitmask(k[INDEXK(rb)].tt) : types[rb];
    bb->t[c].c = ISK(rc) ? bitmask(k[INDEXK(rc)].tt) : types[rc];
    bb->t[c].a = types[ra];
    types[ra] |= bb->t[c].b;
    break;
case OP_FORPREP:
    bb->t[c].a = types[ra];
    bb->t[c].b = types[ra+1];
    bb->t[c].c = types[ra+2];
    types[ra]   = bitmask(LUA_TNUMBER); /* internal index */
    types[ra+1] = bitmask(LUA_TNUMBER); /* limit */
    types[ra+2] = bitmask(LUA_TNUMBER); /* step */
case OP_FORLOOP:
    bb->t[c].a = types[ra];   /* index, limit and step are guaranteed to be numbers */
    bb->t[c].b = types[ra+3]; /* external index */
    types[ra]   = bitmask(LUA_TNUMBER); /* internal index */
    types[ra+3] = bitmask(LUA_TNUMBER); /* external index */
    break;
case OP_NEWTABLE:
    bb->t[c].a = types[ra];
    types[ra] = bitmask(LUA_TTABLE);
    break;
case OP_SETTABLE:
    rb = GETARG_B(i);
    rc = GETARG_C(i);
    bb->t[c].b = ISK(rb) ? bitmask(k[INDEXK(rb)].tt) : types[rb];
    bb->t[c].c = ISK(rc) ? bitmask(k[INDEXK(rc)].tt) : types[rc];
    bb->t[c].a = types[ra];
    break;
case OP_CONCAT:
    rb = GETARG_B(i);
```

```
                        rc = GETARG_C(i);
                        bb->t[c].b = ISK(rb) ? bitmask(k[INDEXK(rb)].tt) : types[rb];
                        bb->t[c].c = ISK(rc) ? bitmask(k[INDEXK(rc)].tt) : types[rc];
                        bb->t[c].a = types[ra];
                        while (rc-rb+1 > 0) {
                            types[rc] = bitmask(LUA_TSTRING);
                            rc--;
                        }
                        types[ra] = bitmask(LUA_TSTRING);
                        break;
                    case OP_GETTABLE:
                        rb = GETARG_B(i);
                        rc = GETARG_C(i);
                        bb->t[c].b = types[rb];
                        bb->t[c].c = ISK(rc) ? bitmask(k[INDEXK(rc)].tt) : types[rc];
                        bb->t[c].a = types[ra];
                    case OP_GETUPVAL:
                    case OP_GETGLOBAL:
                        types[ra] = ~0;
                        break;
                    case OP_VARARG:
                        types[ra] = ~0;
                        break;
                    case OP_SELF:
                        rb = GETARG_B(i);
                        types[ra+1] = types[rb];
                        break;
                    case OP_CLOSURE:
                        bb->t[c].a = types[ra];
                        types[ra] = bitmask(LUA_TFUNCTION);
                        break;
                    case OP_SETGLOBAL:
                    case OP_SETUPVAL:
                        bb->t[c].a = types[ra];
                        break;
                    case OP_JMP:
                    case OP_CALL:
                    case OP_TAILCALL:
                    case OP_RETURN:
                    case OP_TFORLOOP:
                    case OP_TFORPREP:
                    case OP_SETLIST:
                    case OP_CLOSE:
                    default:
                        break;
                }
            }
            /* Check for changes */
            for (c = 0; c < MAXVARS; c++) {
                if (types[c] != old_types[c]) {
                    changes = 1;
                    break;
                }
            }
            memcpy(old_types, types, MAXVARS * sizeof(unsigned int));
        }
    } while (changes);
    free(types);
}
```

Appendix B

Linpack Benchmark in Lua

Here is the Linpack [**?**] benchmark translated into Lua.

```
abs = math.abs

function matgen (a, lda, n, b)
    local norma = 0

    for i = 1, n do
        for j = 1, n do
            a[i][j] = math.random() - 0.5
                norma = (a[i][j] > norma) and a[i][j] or norma
        end
    end

    for i = 1, n do
        b[i] = 0
    end

    for j = 1, n do
        for i = 1, n do
            b[i] = b[i] + a[j][i]
        end
    end

    return norma
end

function run_benchmark (n, ldaa)
    local lda = ldaa + 1
    local a, b, x, ipvt = {}, {}, {}, {}
    for i = 1, lda do
        a[i] = {}
    end
    local cray = .056
    local ops = (2*(n*n*n))/3 + 2*(n*n)

    -- Norm a == max element
    local norma = matgen(a,lda,n,b)

    local time = os.clock()

    -- Factor a
    local info = dgefa(a,lda,n,ipvt)

    -- Solve ax=b
    dgesl(a,lda,n,ipvt,b,0)

    total = os.clock() - time

    for i = 1, n do
        x[i] = b[i]
    end
```

```
    norma = matgen(a,lda,n,b)

    for i = 1, n do
        b[i] = -b[i]
    end

    dmxpy(n,b,n,lda,x,a)

    local resid = 0
    local normx = 0

    for i = 1, n do
        resid = (resid > abs(b[i])) and resid or abs(b[i])
        normx = (normx > abs(x[i])) and normx or abs(x[i])
    end

    eps_result = epslon(1.0)

    residn_result = resid/( n*norma*normx*eps_result )
    residn_result = residn_result + 0.005 -- for rounding
    residn_result = (residn_result*100)
    residn_result = residn_result / 100

    time_result = total
    time_result = time_result + 0.005 -- for rounding
    time_result = time_result*100
    time_result = time_result / 100

    mflops_result = ops/(1.0e6*total)
    mflops_result = mflops_result + 0.0005 -- for rounding
    mflops_result = mflops_result*1000
    mflops_result = mflops_result / 1000
end

function dgefa(a, lda, n, ipvt)
    -- gaussian elimination with partial pivoting

    local info = 0
    local nm1 = n - 1
    if nm1 >= 0 then
        for k = 1, nm1 do
            local col_k = a[k]
            local kp1 = k + 1

            -- find l = pivot index
            local l = idamax(n-k,col_k,k,1) + k
            ipvt[k] = l

            -- zero pivot implies this column already triangularized
            if col_k[l] ~= 0 then

                -- interchange if necessary
                if l ~= k then
                    col_k[l], col_k[k] = col_k[k], col_k[l]
                end

                -- compute multipliers
                local t = -1 / col_k[k];
                dscal(n-(kp1),t,col_k,kp1,1);

                -- row elimination with column indexing
                for j = kp1, n do
                    local col_j = a[j]
                    local t = col_j[l]
                    if l ~= k then
                        col_j[l] = col_j[k];
                        col_j[k] = t;
                    end
                    daxpy(n-(kp1),t,col_k,kp1,1,
                    col_j,kp1,1);
                end
            else
```

```
                info = k
            end
        end
    end

    ipvt[n-1] = n-1
    if a[n-1][n-1] == 0 then info = n-1 end

    return info
end

function dgesl(a, lda, n, ipvt, b, job)
    local nm1 = n - 1

    if job == 0 then
        -- job = 0 , solve  a * x = b.  first solve  l*y = b
        if nm1 >= 1 then
            for k = 1, nm1 do
                local l = ipvt[k]
                local t = b[l]
                if l ~= k then
                    b[l] = b[k]
                    b[k] = t
                end
                local kp1 = k + 1
                daxpy(n-(kp1),t,a[k],kp1,1,b,kp1,1)
            end
        end

        -- now solve  u*x = y
        for kb = 1, n do
            local k = n - (kb - 1)
            b[k] = b[k] / a[k][k]
            local t = -b[k]
            daxpy(k,t,a[k],0,1,b,0,1)
        end
    else
        -- job = nonzero, solve  trans(a) * x = b.  first solve  trans(u)*y = b

        for k = 1, n do
            local t = ddot(k,a[k],0,1,b,0,1)
            b[k] = (b[k] - t)/a[k][k]
        end

        -- now solve trans(l)*x = y
        if nm1 >= 1 then
            for kb = 1, nm1 do
                local k = n - kb
                local kp1 = k + 1
                b[k] = b[k] + ddot(n-(kp1),a[k],kp1,1,b,kp1,1)
                local l = ipvt[k]
                if l ~= k then
                    b[l], b[k] = b[k], b[l]
                end
            end
        end
    end
end

function daxpy(n, da, dx, dx_off, incx, dy, dy_off, incy)
    if n > 0 and da ~= 0 then
        if incx ~= 1 or incy ~= 1 then
            -- code for unequal increments or equal increments not equal to 1
            local ix, iy = 0, 0
            if incx < 0 then ix = (-n+1)*incx end
            if incy < 0 then iy = (-n+1)*incy end
            for i = 1, n do
                dy[iy + dy_off] = dy[iy + dy_off] + da*dx[ix + dx_off]
                ix = ix + incx
                iy = iy + incy
            end
        else
            -- code for both increments equal to 1
```

```
            for i = 1, n do
                dy[i + dy_off] = dy[i + dy_off] + da*dx[i + dx_off]
            end
        end
    end
end

function ddot(n, dx, dx_off, incx, dy, dy_off, incy)
    local dtemp = 0

    if n > 0 then
        if incx ~= 1 or incy ~= 1 then
            -- code for unequal increments or equal increments not equal to 1
            local ix, iy = 0, 0
            if (incx < 0) then ix = (-n+1)*incx end
            if (incy < 0) then iy = (-n+1)*incy end
            for i = 1, n do
                dtemp = dtemp + dx[ix + dx_off]*dy[iy + dy_off]
                ix = ix + incx
                iy = iy + incy
            end
        else
            -- code for both increments equal to 1
            for i = 1, n do
                dtemp = dtemp + dx[i + dx_off]*dy[i + dy_off]
            end
        end
    end
    return dtemp
end

function dscal(n, da, dx, dx_off, incx)
    if n > 0 then
        if incx ~= 1 then
            -- code for increment not equal to 1
            local nincx = n*incx
            for i = 1, nincx, incx do
                dx[i + dx_off] = dx[i + dx_off] * da
            end
        else
            -- code for increment equal to 1
            for i = 1, n do
                dx[i +dx_off] = dx[i +dx_off] * da
            end
        end
    end
end

function idamax(n, dx, dx_off, incx)
    local itemp = 0

    if n < 1 then
        itemp = -1
    elseif n == 1 then
        itemp = 0
    elseif incx ~= 1 then
        -- code for increment not equal to 1
        local dmax = (dx[dx_off] < 0) and -dx[dx_off] or dx[dx_off]
        local ix = 1 + incx;
        for i = 1, n do
            local dtemp = (dx[ix + dx_off] < 0) and -dx[ix + dx_off] or dx[ix + dx_off]
            if dtemp > dmax then
                itemp = i
                dmax = dtemp
            end
            ix = ix + incx
        end
    else
        -- code for increment equal to 1
        itemp = 0
        local dmax = (dx[dx_off] < 0) and -dx[dx_off] or dx[dx_off]
        for i = 1, n do
            local dtemp = (dx[i + dx_off] < 0) and -dx[i+dx_off] or dx[i+dx_off]
```

```
            if dtemp > dmax then
                itemp = i
                dmax = dtemp
            end
        end
    end
    return itemp
end

-- estimate unit roundoff in quantities of size x.
function epslon(x)
    local a = 4.0 / 3.0
    local eps = 0
    while eps == 0 do
        local b = a - 1
        local c = b + b + b
        eps = abs(c-1)
    end
    return eps * abs(x)
end

function dmxpy(n1, y, n2, ldm, x, m)
    -- cleanup odd vector
    for j = 1, n2 do
        for i = 1, n1 do
            y[i] = y[i] + x[j] * m[j][i]
        end
    end
end

run_benchmark(200, 200)

print("Mflop/s:   ",mflops_result)
print("Time:      ",time_result.." secs ("..total.." sec)")
print("Norm Res:  ",residn_result)
print("Precision: ",eps_result)
```

Appendix C

Project Proposal

Jason D. Davies
St John's College
jdd30

Computer Science Tripos Part II Project Proposal

Optimising Lua

Originator: E. C. Upton

22nd October 2004

Project Supervisor: E. C. Upton

Signature:

Director of Studies: E. C. Upton

Signature:

Project Overseers: Dr. N. A. Dodgson & Dr. D. J. Greaves

Introduction

Lua is a simple but powerful programming language widely used for extending applications from a Hazardous Gas Detection System used in the Space Shuttle to online games [1]. It is an extensible extension language [2], which can be embedded in a host program and used as a library of C functions.

There is increasing demand for extension languages as applications become more complex and customization with simple parameters becomes impossible. With its small footprint, simple syntax, portability, safe environment and automatic memory management, Lua is ideal for customising and integrating applications.

Lua is quite fast compared to other interpreted languages, such as Python [3]. However, it is still around 10 times slower than C. The release of Lua version 5.0 saw a 20% speed increase mainly due a switch from a stack-based to a register-based virtual machine [4]. I aim to make the Lua interpreter even faster by implementing a just-in-time (JIT) compiler as well as some program analysis.

The Lua interpreter already pre-compiles chunks of Lua code into an intermediate bytecode to be run in a virtual machine (VM). My JIT compiler will simply compile critical regions of bytecode when needed so that they don't need to be interpreted the next time they are called.

I will also attempt to optimise the interpreter using program analysis. Currently, it seems that executing individual VM instructions is quite expensive because of the dynamic lookups and type checking. If the number of these checks can be reduced, this should produce a noticable performance increase.

Required Resources

As mentioned on the *Project Resource Form* I will use my own PC (2.4GHz Intel Pentium IV with 512Mb RAM and 60Gb hard disc running Linux and Windows XP). Backups will be made daily to the Pelican system, using Subversion for version control, and weekly onto CDs. I will use Intel's VTune profiler [5] to profile the Lua virtual machine.

Starting Point

Lua has been used extensively in commercial environments and is generally considered to be already very stable. The source code is publicly available on the official Lua web site[1]. The Lua 5.0 virtual machine is implemented in around 15,000 lines of ANSI C and uses only 35 opcodes, making this a viable project for the available timescale.

I have a reasonable amount of experience with the C programming language, but I have never used Lua before. My knowledge of compiler and interpreter implementation is limited to the Compilers course taught by Dr. A. Mycroft in Part Ib of the Computer Science Tripos.

Work to be Undertaken

The main project work can be split into milestones as follows.

[1] See `http://www.lua.org/`

1. The **JIT compiler** for Lua. This will compile critical regions of code that are executed a large number of times and may perform other optimisations that exploit information not available to a traditional static compiler. Initially the JIT compiler will simply compile all bytecode without using any heuristics.

2. **Heuristics** for deciding whether the JIT compiler should compile code or let the interpreter execute it on-the-fly. The simplest heuristic and most naïve heuristic is to compile a function on the second call. It is usually faster to interpret a function just once than to compile it and then run the compiled code once. However, a break-even point occurs when the benefits of the optimised, compiled code outweigh the overhead of the initial JIT compilation. We can use the past to try and predict the future, so if a function is called more than once it is likely it will be called again. Sun's Java HotSpot VM uses heuristics to identify "hot spots" that are worth JIT-compiling [6].

3. **Program analysis** with the aim of reducing dynamic lookups and runtime type checking. By caching resolved data types and function table resolutions, these dynamic lookups can be reduced. Runtime type checks could be reduced by supporting additional instructions that operate without type checking. The compiler could be modified to produce these instructions and optimise away dynamic checks based on program context and type inference.

4. **Testing** of the modified interpreter. Test cases will be constructed and used to ensure the optimisations have not introduced errors into the interpreter. Some very simple test cases are already distributed with the Lua source code, but I will need to write some more complex ones to test the interpreter fully.

5. **Benchmarking** of the optimised Lua virtual machine. This will involve writing a benchmarking tool and suitable test cases to demonstrate the optimisations. There are some performance test cases already available on the Web [3].

Some additional areas of interest for further investigation if I have any spare time are detailed below:

- A possible extension is a typed version of Lua, which could be compiled much more efficiently [7]. Dynamic Lua could then be compiled into it, with type checks inserted for values that have to be a particular type (or types). This would need type inference and other tricks in the compiler.

- The JIT compiler will result in more memory usage due to the bulky code it will generate. This problem could be alleviated using mixed-mode JIT compiler. Such a compiler would interpret most instructions but fall back on the interpreter for more complex ones. This is essentially what is done in ARM's Jazelle [8], where most Java bytecode instructions are handled in hardware, but more complex ones are handled in software. A tuning parameter could then adjust the trade-off between speed and space usage. This fallback mechanism could be implemented by providing an extra Lua opcode, which jumps to a block of JIT-compiled assembly code in memory and executes it.

Success Criteria

The following criteria will be used to evaluate the success of my project. The Lua source code includes a number of simple test programs that can be run to help ensure the correctness of my modified interpreter.

1. The new JIT interpreter should execute Lua programs noticably faster than the existing interpreter. This will be verified using the benchmark test cases.
2. The program analysis routines should also result in a noticable speed-up for certain test cases.
3. The optimised interpreter should execute the simple tests bundled with the Lua source code successfully, as well as other selected test cases.

Work Plan

The project will be broken down into the following work packages:

1. **Monday 25th October - Sunday 7th November**
   During the first two weeks I will learn the Lua programming language and familiarise myself with its concepts.
2. **Monday 8th November - Sunday 21st November**
   Become familiar with the source code for the Lua interpreter. Read technical documentation. Identify possible areas for optimisation in the virtual machine.
3. **Monday 22nd November - Sunday 12th December**
   Implement JIT compiler and integrate with Lua source code.

Friday 3rd December End of Michaelmas term.

4. **Monday 13th December - Sunday 19th December**
   Implement advanced heuristics for deciding when to compile code.
5. **Monday 20th December - Sunday 23rd January**
   Implement program analysis routines.

Tuesday 18th January Beginning of Lent term.

6. **Saturday 22nd January - Friday 28th January**
   Test the modified interpreter for correctness. I will run the interpreter on simple test cases to ensure that Lua programs work as expected.
7. **Saturday 29th January - Friday 4th February**
   Write up progress report and submit.

Friday 4th February Progress report submission deadline.

8. **Sunday 5th February - Sunday 27th February**
   Benchmarking and evaluation of the optimised interpreter. I will run both optimised and unoptimised interpreters on test cases and record the times taken to complete each case.

9. **Monday 28th February - Sunday 13th March**
   This is a two week contingency work package to take into account any falling behind e.g. due to a disc failure.

Friday 18th March End of Lent term.

10. **Monday 14th March - Sunday 24th April**
    I have set aside 6 weeks to write up the dissertation. I aim to finish before Easter term starts to avoid disruption from preparing for the written examinations.

Tuesday 26th April Beginning of Easter term.

Friday 22nd May Dissertation submission deadline.

Bibliography

[1] Lua: list of user projects.
http://www.lua.org/uses.html

[2] Ierusalimschy, R. et al. (1996). Lua: an extensible extension language. *Software: Practice & Experience* 26, 6, pp 635–652.

[3] The Great Computer Language Shootout.
http://shootout.alioth.debian.org/, 22 Oct 2004.

[4] Ierusalimschy, R. (Aug 2002). Re: virtual machine.
http://lua-users.org/lists/lua-l/2002-08/msg00198.html

[5] Intel's VTune Profiler.
http://www.intel.com/software/products/vtune/

[6] Java HotSpot Technology.
http://java.sun.com/products/hotspot/

[7] Thomas, R. (Mar 2001). Re: JIT for Lua.
http://lua-users.org/lists/lua-l/2001-03/msg00036.html

[8] ARM Jazelle Technology.
http://www.arm.com/products/solutions/Jazelle.html