Tecnical Documentation

trace-based just-in-time compiler

LuaJIT

Authors
Dario d'Andrea
Laurent Deniau

April 22, 2020

Preface

This document describes the details of the LuaJIT project with a particular focus on the concept of trace-based just-in-time compilation. It aims to be a general guide especially for newcomers of this topic. This document takes its birth from a recent Master thesis project [1] that was carried out at CERN, the European Organisation for Nuclear Research.

Before to start, we want to express our gratitude to the author of LuaJIT, Mike Pall, and to its community for all the useful insights contained in the LuaJIT mailing list.

Contents

Chapter 1

LuaJIT overview

This chapter aims to present an overview of LuaJIT. The first paragraph outlines the Lua programming language, then the following paragraphs focus on LuaJIT architecture, virtual machine (VM) and just-in-time compiler (JIT).

1.1 Lua

Lua is described on its official website [2] as a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. It is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua runs by interpreting bytecode with a register-based virtual machine. It is dynamically typed and has automatic memory management with incremental garbage collection.

Lua is specifically known for its performance. Experiments on several benchmarks show Lua as one the fastest interpreted scripting languages ever made.

1.2 LuaJIT

LuaJIT [3] is a trace-based just-in-time compiler for the Lua programing language. It is widely considered to be one of the fastest dynamic language implementations as it outperforms other dynamic languages on many cross-

language benchmarks. In this paragraph, we will go through a description of its internal architecture, which is shown in Fig. 1.1.

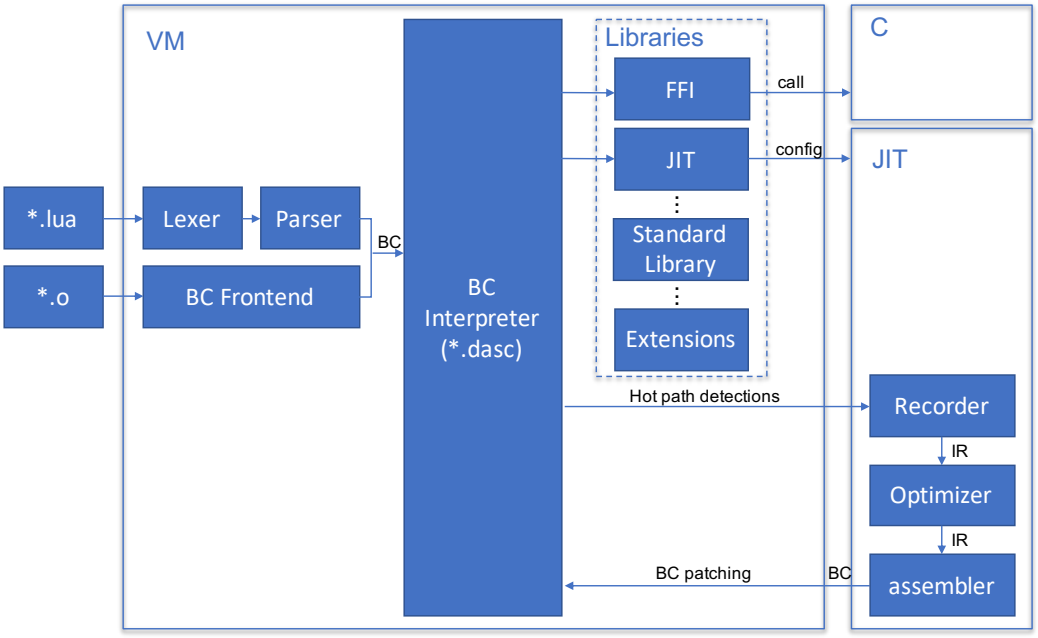

Figure 1.1: Schematic view of LuaJIT internals (source [4])

The compiler fronted is represented by the blocks *Lexer*, *Parser* and *BC Frontend*. The input can either be text files containing Lua code (`*.lua`) or object files (`*.o`). Lua files are processed by *Lexer* and *Parser* that generate LuaJIT bytecode instructions (BC). On the other hand, object files, which contain already converted LuaJIT bytecode instructions, are read by the *BC Frontend*.

Once the source code is translated to bytecode instructions, the *BC interpreter* executes them. The interpreter of LuaJIT is fully written in assembly (files `*.dasc`). It performs very fast interpretation and it uses DynASM [5] as a powerful cross-platform assembler.

Several libraries are used to support the *BC interpreter* in its tasks. Within these library the most relevants are: (i) the foreign function interface (*FFI*) which allows to inject C code into Lua in a very efficient way; (ii) the *Standard Library* to manipulate Lua data (stack, registry, etc.); (iii) the *JIT* library that provides the functions to configure the JIT engine; (iv) the *Extentions* of Lua.

The core of LuaJIT is represented by the JIT engine. LuaJIT detects hotpaths from frequently executed fragments of code. Once a hotpath is

detected, the executed bytecode instructions are recorded into a trace by the *Recorder* which also emits an intermediate representation (IR) in Static single assignment (SSA) form. Then, the *Optimiser* applies some optimisations on the IR followed by the *Assembler*, which compiles the trace to platform-specific machine code. Finally, the bytecode that was detected as hotpath is patched in order to replace its execution with a call to the compiled trace. The details of the just-in-time compilation are described at chapter 4.

1.3 Files organisation

LuaJIT consists of several files which are listed in the table below grouped by similar purposes. It should be noted that the files used to build LuaJIT are excluded from this table.

File	Description
`luaconf.h`	Lua configuration header
`lua.[h,hpp]`	Lua
`lauxlib.h`	Auxiliary functions for building Lua libraries
`lib_aux.c`	Auxiliary library for the Lua/C API
`lualib.h`	Standard library header
`lib_base.c`	Base and coroutine library
`lib_math.c`	Math library
`lib_string.c`	String library
`lib_table.c`	Table library
`lib_io.c`	I/O library
`lib_os.c`	OS library
`lib_package.c`	Package library
`lib_debug.c`	Debug library
`lib_bit.c`	Bit manipulation library
`lib_jit.c`	JIT library
`lib_ffi.c`	FFI library
`lib_init.c`	Library initialization
`Makefile`	LuaJIT Makefile
`ljamalg.c`	LuaJIT core and libraries amalgamation
`luajit.[c,h]`	LuaJIT frontend
`lj_alloc.[c,h]`	Bundled memory allocator
`lj_api.c`	Public Lua/C API
`lj_arch.h`	Target architecture selection
`lj_buf.[c,h]`	Buffer handling
`lj_def.h`	LuaJIT common internal definitions

lj_ff.h	Fast function IDs
lj_ffrecord.[c,h]	Fast function call recorder
lj_frame.h	Stack frames
lj_func.[c,h]	Function handling (prototypes, functions and upvalues)
lj_gc.[c,h]	Garbage collector
lj_gdbjit.[c,h]	Client for the GDB JIT API
lj_obj.[c,h]	LuaJIT VM tags, values and objects
lj_lib.[c,h]	Library function support
lj_load.c	Load and dump code
lj_udata.[c,h]	Userdata handling
lj_carith.[c,h]	C data arithmetic
lj_ccall.[c,h]	FFI C call handling
lj_ccallback.[c,h]	FFI C callback handling
lj_cconv.[c,h]	C type conversions
lj_cdata.[c,h]	C data management
lj_char.[c,h]	Character types
lj_clib.[c,h]	FFI C library loader
lj_cparse.[c,h]	C declaration parser
lj_crecord.[c,h]	Trace recorder for C data operations
lj_ctype.[c,h]	C type management
lj_str.[c,h]	String handling
lj_strfmt.[c,h]	String formatting
lj_strfmt_num.c	String formatting for floating-point numbers
lj_strscan.[c,h]	String scanning
lj_tab.[c,h]	Table handling
lj_bcdef.h	Generated file
lj_bc.[c,h]	Bytecode instruction format and modes
lj_bcdump.h	Bytecode dump definitions
lj_bcread.c	Bytecode reader
lj_bcwrite.c	Bytecode writer
lj_lex.[c,h]	Lexical analyzer
lj_parse.[c,h]	Lua parser (source code -> bytecode)
lj_vm.h	Assembler VM interface definitions
lj_vmevent.[c,h]	VM event handling
lj_vmmath.c	Math helper functions for assembler VM
vm_arm.dasc	Low-level VM code for ARM CPUs
vm_arm64.dasc	Low-level VM code for ARM64 CPUs
vm_mips.dasc	Low-level VM code for MIPS CPUs
vm_mips64.dasc	Low-level VM code for MIPS64 CPUs

vm_ppc.dasc	Low-level VM code for PowerPC 32 bit or 32on64 bit mode
vm_x64.dasc	Low-level VM code for x64 CPUs in LJ_GC64 mode
vm_x86.dasc	Low-level VM code for x86 CPUs
lj_jit.h	Common definitions for the JIT compiler
lj_trace.[c,h]	Trace management
lj_traceerr.h	Trace compiler error messages
lj_dispatch.[c,h]	Instruction dispatch handling
lj_ir.[c,h]	SSA IR (Intermediate Representation) format and emitter
lj_ircall.h	IR CALL* instruction definitions
lj_record.[c,h]	Trace recorder (bytecode -> SSA IR)
lj_snap.[c,h]	Snapshot handling
lj_state.[c,h]	State and stack handling
lj_iropt.h	Common header for IR emitter and optimizations
lj_opt_dce.c	Dead Code Elimination. Pre-loop only (ASM already performs DCE)
lj_opt_fold.c	Constant Folding, Algebraic Simplifications and Reassociation. Array Bounds Check Elimination. Common-Subexpression Elimination.
lj_opt_loop.c	Loop Optimizations
lj_opt_mem.c	Memory access optimizations. Alias Analysis using high-level semantic disambiguation. Load Forwarding (L2L) + Store Forwarding (S2L). Dead-Store Elimination.
lj_opt_narrow.c	Narrowing of numbers to integers (double to int32_t). Stripping of overflow checks.
lj_opt_sink.c	Allocation Sinking and Store Sinking
lj_opt_split.c	Split 64 bit IR instructions into 32 bit IR instructions
lj_mcode.[c,h]	Machine code management
lj_meta.[c,h]	Metamethod handling
lj_emit_arm.h	ARM instruction emitter
lj_emit_arm64.h	ARM64 instruction emitter
lj_emit_mips.h	MIPS instruction emitter
lj_emit_ppc.h	PPC instruction emitter
lj_emit_x86.h	x86/x64 instruction emitter
lj_target.h	Definitions for target CPU
lj_target_arm.h	Definitions for ARM CPUs

lj_target_arm64.h	Definitions for ARM64 CPUs
lj_target_mips.h	Definitions for MIPS CPUs
lj_target_ppc.h	Definitions for PPC CPUs
lj_target_x86.h	Definitions for x86 and x64 CPUs
lj_asm.[c,h]	IR assembler (SSA IR -> machine code)
lj_asm_arm.h	ARM IR assembler (SSA IR -> machine code)
lj_asm_arm64.h	ARM64 IR assembler (SSA IR -> machine code)
lj_asm_mips.h	MIPS IR assembler (SSA IR -> machine code)
lj_asm_ppc.h	PPC IR assembler (SSA IR -> machine code)
lj_asm_x86.h	x86/x64 IR assembler (SSA IR -> machine code)
lj_debug.[c,h]	Debugging and introspection
lj_err.[c,h]	Error handling
lj_errmsg.h	VM error messages
lj_profile.[c,h]	Low-overhead profiling

Table 1.1: LuaJIT files

Chapter 2

Background

2.1 Just-in-time compilation

Over the years the research community in computer science has tackled the problem of improving dynamic languages performances using different approaches, which can be classified in two major categories [6, 7]: (i) writing fast interpreters; (ii) integrating interpreters with just-in-time compilers (JIT).

Differently from static compilers, interpreters inevitably introduce overhead caused by running the interpreter itself. Thus, when writing fast interpreters the goal is to reduce at minimum the cost of interpretation overhead. In fact, implementing fast interpreters is proved to be effective only for languages where the cost of interpretation dominates the total execution time.

On the other hand, JIT compilers try to optimise different kinds of overhead introduced by dynamic languages. In contrast to ahead-of-time compilation, where a program is statically compiled upfront and then run, just-in-time compilation is a technique where the machine code is emitted at run-time according to the observed program's execution. As a result of delaying compilation at run-time, the JIT can take into account specific features of the program's execution when generating the machine code. In this way, it can perform more aggressive optimisations.

JIT compilers are usually applied in the context of interpreted-based system where a program is represented in the form of bytecode executed by a virtual machine (VM). In this case a program is interpreted at first by the VM, then the JIT compiles only frequently executed parts of the code defined as *hotspots*. These are the parts where the program spends most of its time, hence emitting efficient machine code should naturally lead to improving the overall performance.

Cuni in [7] illustrates two general rules to consider when tackling the problem of compiler optimisation: (i) the *Pareto principle* (or *80/20 rule*) [8] states that the 80% of the execution time of a program is related only to 20% of the code. Thus, small parts of the code can make the difference in the performance of the whole program; (ii) the *Fast Path principle* [9] explains that the most frequently used operations should be handled by *fast paths* in order to speed up the execution, while the remaining cases are not required to be particularly efficient.

2.2 Compilation units

A key designing decision for JITs is to define what constitutes the basic compilation unit, which in a classical compilers approach is represented by a whole file or module. In the contest of just-in-time compilation, considering such a large component would not give the advantages expected because it can cause a substantial delay in programs execution. In this case, smaller compilation units, which refers only to most frequently executed parts of the code (*hotspots*), are more adequates. This choice will also decrease memory usage minimising the total amount of compiled code.

Schilling in [10] illustrates common choices of compilation units used over the years in the context of dynamic optimisation systems:

(i). *Dynamic Basic Block*. As defined by Smith and Nair [11] a dynamic basic block is determined by the actual flow of a program when it is executed. It always begins at the instruction executed immediately after a branch and it continues until the first next conditional branch is encountered. Dynamic basic blocks are usually larger than static basic blocks and the same static instruction may belong to more than one dynamic basic block. This approach is typically used in binary translators.

(ii). *Function* (*Method*). It is the most intuitive compilation unit for a JIT compiler. In this case, the whole function with all the possible branches and control flow paths is compiled. A function is generally marked as hot and compiled when it is frequently called at run-time. Then, any subsequent calls of the same function will lead to the already compiled machine code, instead of using the interpreter. Afterwards, the system generally reverts to interpretation when the compiled function ends. Also static compilers usually compile a function all at once, hence the same optimisation techniques can be used for function-based just-in-time compilers.

(iii). *Loop*. The analogous approach used for functions can be applied for loops. In this context the entire loop body is compiled, including all possible control-flow paths. Loops are generally good candidates to be considered as hotspots since the same set of instructions will be executed repeatedly many times.

(iv). *Region*. Firstly introduced in [12], this approach uses regions as more general compilation units. A region is the result of collecting code from several functions, but it excludes all rarely executed portions of these functions. To create a region the process begins by the most executed block not yet in the region, so-called *seed block*. Then, the scope of the region is expanded by selecting a path of successors based solely on the execution frequency. This process continues until no more desirable successors are found.

(v). *Trace*. A trace is a linear sequence of instruction that does not contain any control-flow joint points. The execution either continues on the trace, which consists of a unique path of instructions (*hotpath*), or it exits the trace. A trace can have a single entry point and one or more exit points. According to the logic used in designing the JIT, traces can be generated from loops or functions. The last instruction of the trace may jump to the beginning of the trace (e.g. loops) or to another trace or to the interpreter. Trace exits can either lead to another trace (*sidetrace*) or back to the interpreter. If there are multiple frequently executed control flow paths related to the same set of instructions, the JIT will generate multiple traces (including sidetraces). This can lead to duplication because a block of instruction can be repeated in different traces, but this replication can provide more opportunities for specialisation and aggressive optimisation.

An interesting study by Bruening and Duesterwald [13] investigates strategies for finding the optimal compilation unit shapes. They show that the hybrid combination of functions with traces or loops significantly outperforms the solely function-based strategy.

In the following section, we will go through an extensive explanation of trace-based just-in-time compilers.

2.3 Trace-based Just-in-time Compilation

A JIT compiler that considers *traces* as compilation unit is called *trace-based just-in-time compiler* or *tracing JIT*. Frequently executed fragment of code

(either loops or functions) are good candidates to produce hotpaths that will be compiled into traces.

This family of just-in-time compilers is built on the assumptions that: (i) programs spend most of their execution time in loops; (ii) several iterations of the same loop are likely to take similar code paths.

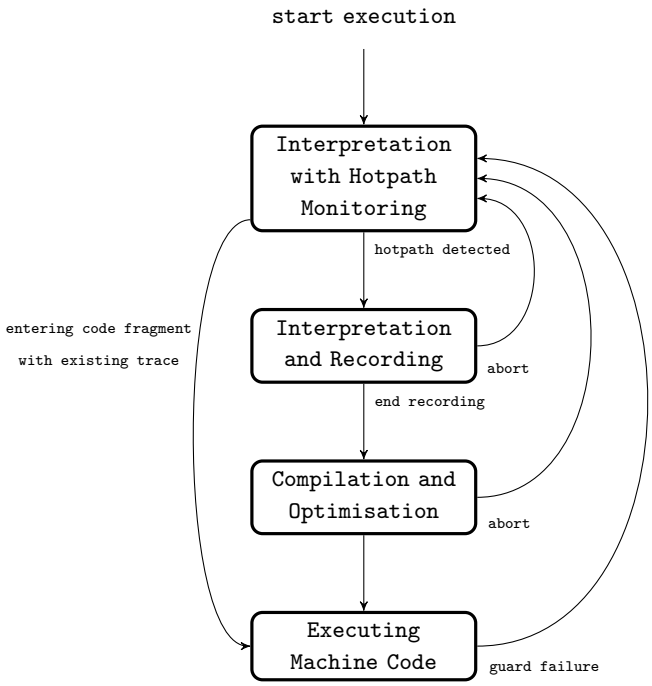

Figure 2.1: Stages of execution for a VM with tracing JIT

A system made by a virtual machine (VM) equipped with a tracing JIT can go through various stages when executing a program. These are summarised in Fig. 2.1:

(i). *Interpretation.* At first the program is run by the interpreter, which executes bytecode instructions while performing some light profiling in order to identify hotpaths of the program. In particular, it monitors the bytecode instructions that may be potential *trace headers*, which are the instructions from where a trace can start. The techniques used to monitor potential trace header and to identify hotpaths can vary for different implementations of tracing JITs (see Sections 2.3.1, 2.3.2). Most of them use a counter that is incremented every time a potential trace header instruction is executed. When the counter exceeds a certain threshold the VM switches to recording mode.

On the other hand, if the interpreter hits a fragment of code that has already been compiled into an existing trace, the execution goes to the already compiled machine code. In this case the VM switches to executing machine code.

(ii). *Recording.* When a hotpath is found the interpreter continues to run bytecode instructions, but all the executed bytecode instructions are also recorded. These recorded instructions are stored into a linear list that we previously called *trace*. Generally, from the bytecode instructions it is emitted an intermediate representation (IR) that will be used for optimisation and compilation.

Recording continues until the interpreter finishes to execute all the instructions of the detected hotpath (e.g. one iteration of a loop or an entire function). The decision to stop recording is crucial for the efficacy and performance of a tracing JIT. It will be further discussed in Section 2.3.3.

At any moment of the recording phase, an abort can occur. It means that recording failed because the execution flow took a path in the code that cannot produce a suitable trace. This can be caused by an exception or any kind of error while recording. If this happens the partial trace that was generated is discarded and the VM switches back in the previous stage that consists of sole interpretation.

(iii). *Optimisation and Compilation.* Once recording is successfully completed the system switches to compilation. In this case the IR produced is aggressively optimised and the trace is compiled to machine code. The JIT compiler produces very efficient machine code that is immediately executable, e.g. it can be used for the next iteration of a loop or for the next time a function will be called.

During this phase an abort can also occur. If so, the partial trace is discarded and the system switches back to interpretation.

(iv). *Executing Machine Code.* In this phase the machine code previously generated by the tracing JIT is executed. This machine code is cached so that if the interpreter encounters a code fragment that previously produced a trace, it will switch to executing the already compiled machine code. Generally, there is a limited cash memory where compiled traces are stores; if this memory is full the oldest trace is discarded to give place for the new one.

The end of a trace can either be connected: (1) to itself (i.e. loop or recursive function), thus the machine code of the trace runs repeatedly

until some exit condition is triggered; (ii) to another trace; (iii) to the interpreter. This link is created according to the specific hotpath previously recorded and executed by the interpreter.

Since a trace is a linear sequence of instructions, it contains *guards* that ensure the correctness of the machine code executed. Guards check that the assumptions in the trace are fulfilled (e.g. the execution flow follows a specific path of a branch, the hypothesis on variables types are verified, etc). If one of the assumptions is not respected the associated guard fails and the trace exits. When the trace exits because of a guard failure the system generally switches to interpretation, but if particular conditions are met (see Section 2.3.7) a trace exit can lead to another trace, so-called *sidetrace*.

In the next sections, we will describe the phases just mentioned more in details.

2.3.1 Identifying trace headers

Identifying bytecode instructions that are potential trace headers is a key and delicate aspect for trace-based just-in-time compilation. This task must be very effective because we want to select only code paths where the program actually spends a lot of time. On the other hand, it is desirable to reduce at minimum the activity of monitoring during interpretation because it may impact the performances.

Schilling in [10] discusses different methods adopted in literature for identifying potential trace headers:

(i). *Next executed tail (NET).* This is the first and most intuitive method used to identify hotpaths in programs. Introduced by Bala, Duesterwald, and Banerjia in [14, 15], it is based on the assumption that every loop must contain at least one backward branch, which is a jump to a lower address in memory with respect to the current program counter. The target of the backward branch is considered as a potential trace header because it is the first instruction of the loop. With this rationale, the target instruction of function calls could also be considered as a potential trace header when there is a backward branch. Many tracing JITs adopt this heuristic e.g. HotpathVM [16] and PyPy's Tracing JIT Compiler [17].

(ii). *Last Executed Iteration (LEI).* This method, introduced by Hiniker, Hazelwood, and Smith in [18], is a specialisation of NET. It also considers only the targets of backward branches as potential trace headers,

but it keeps track of the last n branch targets in a history cache. Only branch targets in this cache will be considered as potential trace header. Even if this method implies an overhead caused by the cache, it needs fewer counters because there will be fewer branch targets. Hiniker, Hazelwood, and Smith proved that using LEI (instead of NET) there is an improvement in locality of execution while reducing the size of the code cache.

(iii). *Natural loop first (NLF).* This approach consists in considering some bytecode instructions as "special" because they are the only ones that can be potential trace headers (e.g. bytecode instructions at the beginning of a loop, or function call). A special treatment should also be performed for recursive functions and gotos that can rise with high probability frequently executed paths in the code. To use this technique we must be able to access the information on the higher-level structure of the program. The advantage of this method is that fewer points of the program are considered as potential trace headers and fewer counters are needed. It is also more predictable to know where traces can start.

LuaJIT [3] by Pall uses in fact this heuristic to identify hotpaths, e.g. a `for` is translated to a special bytecode instruction that is considered as potential trace header.

It should be noted that side exits of a trace can also be considered as potential trace headers because a trace, that we previously called sidetrace, can start from that point (paragraph 2.3.7 describe this technique in details).

2.3.2 Hotpath detection

Once the interpreter identifies a bytecode instruction that is a potential trace header (using whatever techniques previously described), a counter associated to that fragment of code is incremented. Finally, the tracing JIT detects a hotpath when the counter exceeds a certain threshold (*hotness threshold*). Fig. 2.2 shows a diagram that explain this mechanism.

The value of hotness threshold is a critical aspect that can indeed affect the performances of a tracing JIT. Having a low hotness threshold implies that fragments of code that are not actually "hot" can be compiled. In this case compiling that fragments was not worth it because it only brought compilation overhead. On the other hand, a high hotness threshold can imply that the execution flow stays too much time in the interpreter and the system does not exploit the advantages of compiling frequently executed fragments

of the code. Finding a suitable trade-off depends on many aspects including the specific application, programming language, architecture, etc.

In many tracing JIT, the hotness threshold is a parameter that the user can set according to its needs. In this way it is possible to change it based on the performances obtained.

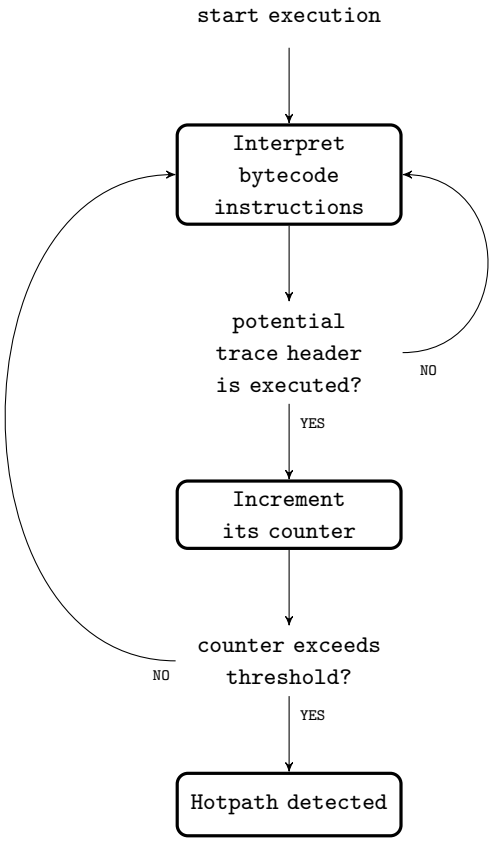

Figure 2.2: Hotpath detection

2.3.3 Trace recording

As described before, recording starts when a hotpath is detected. The interpreter switches to "recording mode" so it will interpret and record into a trace the executed instructions. A trace is entirely specialised on the path that the execution flow takes when recording instructions. Specialisation is a key aspect for tracing JITs because the final goal is to create very efficient and specialised machine code. However, the recording technique is very speculative because there are no guarantees on which path the execution flow will

take when recording instructions. Ideally, we should record the path that has the highest probability to be taken, but this is not ensured in any way.

Analysing the example of the loop in Fig. 2.3 clarifies this concept. The two possible paths taken by the execution flow are either `A-B-D` (path 1) or `A-C-D` (path 2) since there is a branch after the block `A`. Let's suppose that, in a random iteration of the loop, the probability of executing path 1 is 80% and the remaining 20% for path 2. In this situation the best would be to record the trace considering path 1, but there is no guarantee of that. In fact, the behaviour of a tracing JIT is the following. As usual the program is run by in the interpreter at first, then VM starts recording when the counter exceeds the hotness threshold (assuming that the loop iterates enough time to become hot). The path that will be recorded is the path taken by the execution flow in the next iteration of the loop when the system switches to mode "interpretation and recording" (it can be either path 1 or path 2). Assuming that we are not unlucky path 1 will be recorded, but there is no guarantee of that.

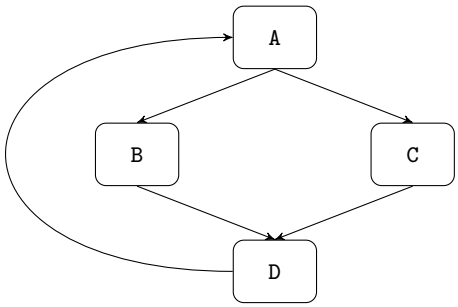

Figure 2.3: Example of loop

The phase of interpretation and recording may continue until either there is an abort or an end-of-trace condition is met, which means that the recording is successfully completed (abort will be discussed in the next paragraph). In a successful scenario, a tracing JIT stops recording because one of the following end-conditions has been encountered [10]: (i) *Loop back to entry.* It is the most simple case when a loop goes back to the point where recording started. It means that a cycle has been found, thus recording can stop because a loop has been correctly recorded; (ii) *Loop back to parent.* It is the case when a sidetrace loops back to its parent trace. Thus, a cycle has been detected and the sidetrace was successfully recorded; (iii) *Start of existing trace.* This happens when an already existing trace is encountered while recording. In this situation the behaviour of a tracing JIT can vary for different implementations: either it stops recording and the trace jumps

to the existing trace or recording will continue independently. In the latter situation there will be longer traces and duplication increases, but there are more opportunities for specialisation and aggressive optimisations.

2.3.4 Abort and blacklisting

Trace abort can happen for multiple reasons at any stage of trace creation (i.e. recording, optimisation or compiling). If an exception is thrown while recording, the trace is aborted because this represents an exceptional (and usually rare) program state. Similarly, certain very expensive instructions cause trace abort because their cost exceeds the potential run-time saving that can be realised through compiling that code fragment to machine code (e.g. memory allocation instructions). Another possible cause of abort is an overlong trace which means that recording is aborted when the trace becomes too long. Hypothetically, the entire program could be covered by a single trace, at the expenses of having an inevitably huge trace. This is clearly not our goal because it will not lead to any benefits. Finally, another common situation that causes trace abort is when we try to record a bytecode instruction that cannot be translated into machine code because the tracing JIT does not support this feature. This can happen either because the feature was not yet implemented or because who designed the tracing JIT voluntarily decided not to support it for any reason (e.g. there was no advantage in compiling traces that contains this instruction).

There could be a scenario where a tracing JIT repeatedly tries to create a trace from a fragment of code (either a loop or function), but trace creation always aborts. In this case the interpreter spends times trying to record traces, but it will never be able to create any. Thus, a simple technique to adopt in this situation is to blacklist traces that failed to compile many times. Through a counter, so-called *backoff* counter by Gal et al. [19], the number of recording attempts is bounded to a certain limit. If the number of failed attempts of recording a trace from a code fragment exceeds this limit, the fragment is blacklisted and the interpreter will never retry to start recording at that point again. Fig. 2.4 describes this mechanism.

Some tracing JITs (e.g. LuaJIT [3]) adopt the policy that a blacklisted fragment of code cannot be whitelisted ever again. While other implementations (e.g. RaptorJIT [20]) give another chance to a blacklisted fragment because sometimes code fails to compile in isolation, but the same code can be compiled in a different context.

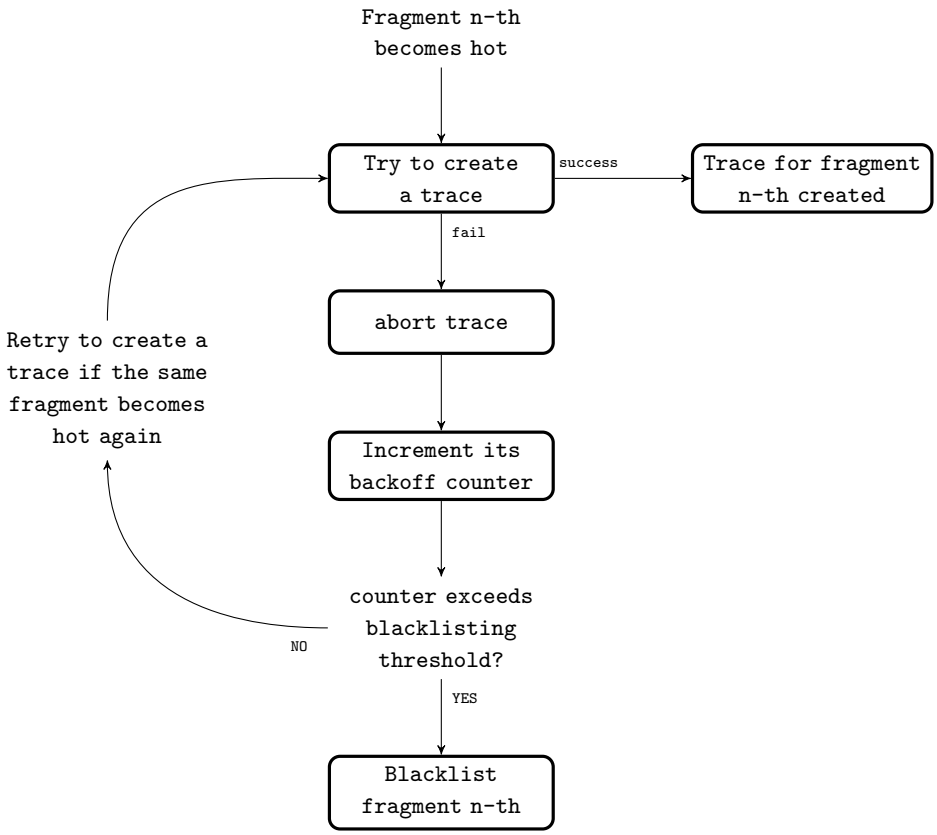

Figure 2.4: Blacklisting

2.3.5 Compiling traces

As explained previously, in "interpreting and recording" mode the interpreter executes bytecode instructions and records them into a trace. In that phase it is usually also emitted an intermediate representation (IR), which is in Static Single Assignment (SSA) form [21]. Each variable is assigned exactly once, and every variable is defined before it is used. Gal et al. introduced in [16] a novel form of SSA, so-called Trace Static Single Assignment (TSSA) which exploit the fact that traces only follow exactly one path.

Before producing the actual machine code, a trace is optimised. In fact, since traces do not contain multiple control flows, it is simple to apply optimisations on a linear set of instructions. Tracing JITs use most of the well-known compiler techniques, e.g. constant propagation, constant folding, redundant guard removal, store/load propagation, allocation removal, common sub-expression elimination, dead code elimination, loop invariant code motion, loop unrolling, code sinking.

After optimisations, a trace is compiled to very efficient and specialised machine code where every guard is turned into a quick check to verify whether the assumption still holds. At this point, the trace consists of a linear sequence of optimised instructions in SSA form, hence the translation to machine code is also facilitated.

2.3.6 Trace exit

A trace is executed linearly from its first instruction to the last assuming that all the assumptions are respected by success of all guards. As mentioned previously, the end of the trace can either be connected: (1) to itself (i.e. loop or recursive function), thus the machine code of the trace runs repeatedly until some exit condition is triggered; (ii) to another trace; (iii) to the interpreter. If the assumptions checked by the guards are not verified the trace exits.

When the execution leaves a trace because of a guard failure the system switches back to interpretation. The VM should be left in a consistent state for the interpreter to continue. In particular, the values held in registers throughout the trace must be written back to their respective stack locations. Once the stack is in a suitable state, the interpreter can continue.

A naive solution to this problem could be to force a full update of the state to memory before every exit. However, this solution seriously decrements code performance.

A better approach introduced in [14] accomplish this task with the so-called *exit stubs*. They consist of small pieces of code that execute the necessary writes. With this approach a guard is implemented as a conditional branch to the exit stub when it fails. At the end of an exit stub, there is a jump to a routine that transfers control to the interpreter. Since for some architectures conditional branches have a limited jump distance, the code responsible for exits stub is often located just after the trace. Many tracing JITs use exit stubs to keep the VM state consistent because they proved to be very efficient. However, they imply some drawbacks: (i) there is an overhead because we need to produce extra code (ii) they may cause fragmentation of the machine code area. If a sidetrace is attached to an exit, the exit stub is no longer needed and its memory can be used for other purposes [10].

An alternative technique is to save the contents of all registers on trace exits and use meta-data stored with the trace to recover a consistent state. This approach is used in LuaJIT [3] with *snapshots* that store a consistent view of all updates to the state before an exit. This data-driven approach is slower if compared to exit stubs, but it avoids the need of generating extra code. This slowness does not have a serious impact on the performances because repeatedly taken exits generate sidetraces. Trace exits that go back

to interpretation should be relatively rare events.

2.3.7 Sidetraces

As previously mentioned, a trace can be created from an exit of a root trace. The trace generated will be called *sidetrace* because it starts from the exit of another trace. The trace to which the sidetrace is attached is called *parent trace*. Sidetraces are needed because a single trace only covers one path of the entire control flow graph. If multiple paths become hot (in the sense of being frequently executed) it is appropriate to compile them in multiple traces.

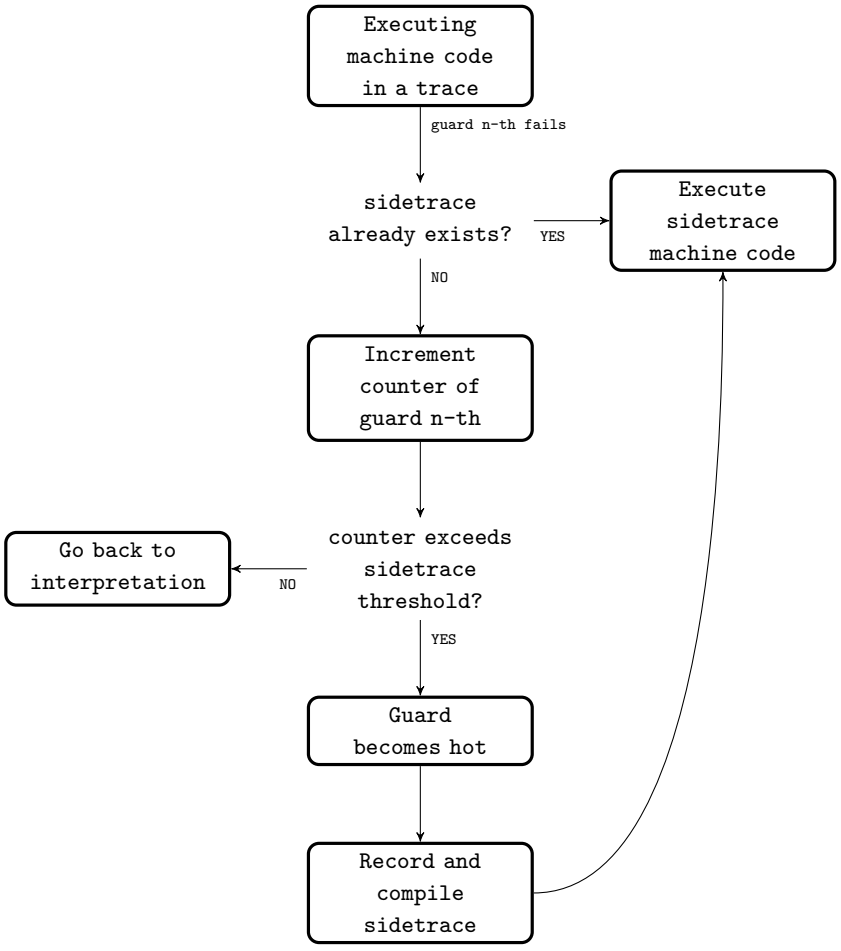

Figure 2.5: Sidetrace creation

A sidetrace is created when the same guard fails repeatedly (the guard becomes hot). At that point, it is too expensive to restore the VM and to

resume interpretation. Thus, it is more profitable to attach a sidetrace to the hot exit. The diagram in Fig. 2.5 describe this mechanism.

In a situation where two paths are frequently executed, the first path that becomes hot will be handled by the parent trace, then the second one will be handled in part by the parent trace and finally by the sidetrace. The example of the loop in Fig. 2.6 describes this situation. If the path `A-B-D` becomes hot first a root trace (trace with no parent) will be created. Then a sidetrace that executes `C-D` is created when the guard also becomes hot.

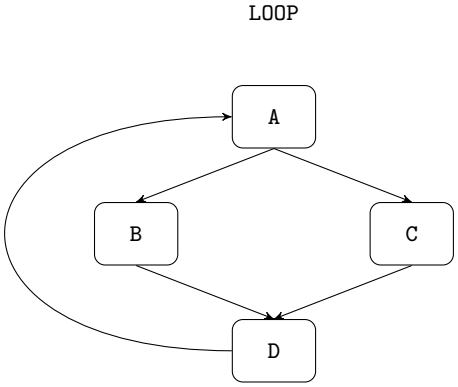

Figure 2.6: Loop

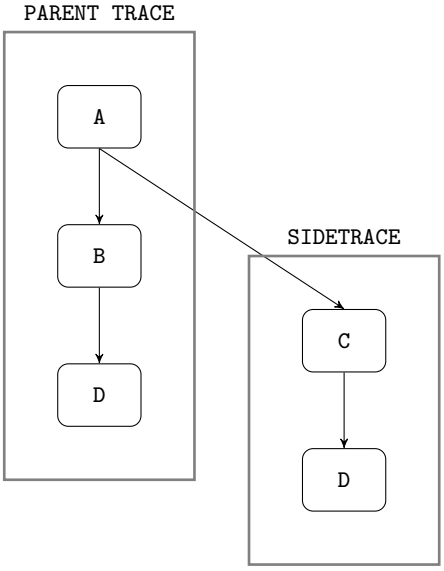

Figure 2.7: Traces generated

The creation of a sidetrace must be efficient to not deteriorate the overall performances. When creating a sidetrace there is a drawback if the values of the registers in the parent trace are first written back to the stack in order to be read back in the new sidetrace. The best would be to transfer directly the values from parent trace to sidetrace.

A possible solution consists in *trace trees*, introduced by Gal and Franz [22], where the unit of compilation is a root trace with all its attached sidetraces. This technique can profit from more aggressive optimisations on the entire trace tree, but it needs to recompile the whole tree when a new trace is attached.

Another approach, which is used in LuaJIT [3], is called *trace coalescing*. The trace compiler maintains a mapping between register and stack slots that is used in the compiler for the sidetrace. The tracing JIT does not emit a load from the stack slot, but it emits a read from the registers that have

the contents of the stack slot in the parent trace.

2.4 Related works

Before to go more in the details of LuaJIT we present an historical overview of other projects related to trace-based just-in-time compilation.

2.4.1 Early Tracing JITs

The idea of tracing just-in-time compilation was first introduced in 1970 by Mitchell [23]. He observed that programs can be compiled at run-time by simply storing the actions performed during interpretation. The compiled code can be derived by executed program actions since it is likely to remain valid and usable for a reasonable time. If that code ever become invalid due to a change in any of its assumptions, the system should be able to revert to interpretation.

While Mitchell was the first to introduce the concept of tracing just-in-time compilation, the first widely known system using this approach was Dynamo by Bala et. al [24] in 2000. It is a framework for dynamic routine optimisation of binary code that records frequently executed traces and optimises instruction in that trace. They pioneered the technique of compiling only partial parts of the code classified as "hot".

As described by Aycock [25], other projects were developed along with Dynamo. They were focused on CPU emulation, which is a dynamic binary translation of paths, or traces, that involves translating machine codes from one architecture to another at run time. The most dominants in literature refers to Deaver et al. [26] in 1999, Gschwind et al. [27] in 2000, Zheng et al. [28] in 2000. These early compilers differ from later tracing JITs which usually work on a higher level (either on bytecode or intermediate representation level). However, they introduced the key concept of tracing just-in-time compilation: the compilation unit consists of "hot" program paths, or traces, rather than methods, as it was usually done by previous JIT compilers. A path reflects the control flow exhibited by the source program at run-time, a dynamic instead of a static unit of translation.

Further work was done in 2003 by Sullivan et al. [29]. They implemented from Dynamo a new tracing JIT compiler called DynamoRIO. It introduced the concept of *meta-tracing* where the JIT compiler does not trace the user program being run, but it traces the execution of the interpreter while it runs this program.

2.4.2 Recents Tracing JITs

In more recent years, several tracing just-in-time compilers have been proposed as an efficient solution for dynamic languages. HotpathVM by Gal et al. [16] is a tracing JIT for Java VM released in 2006. It is small enough to fit on resource-constrained embedded devices. It dynamically builds traces from the bytecode and it limits its effort to frequently executed loops that are identified through backward branches. The key of their success consists in an innovative use of the SSA [21] transformation, which Gal et al. called TSSA (Trace Static Single Assignment). In classical SSA a control-flow graph is entirely transformed into SSA form, and ϕ nodes are placed in control-flow merge points. TSSA consists in transforming into SSA form only variables that are actually used in a recorded trace. In this way it is possible to perform more aggressive optimisations on the trace including LICM (loop invariant code motion) and moving operations on SSA values across side exit points.

Later on, in 2009, Gal et al. applied their trace-based approach to JIT compilation of dynamically-typed languages. They developed a tracing JIT compiler for JavaScript, so-called TraceMonkey [19]. It was implemented for an existing JavaScript interpreter called SpiderMonkey [30] and it was used in Mozilla's Firefox Browser up to version 11 of Firefox. TraceMonkey proposed a novel approach including trace trees. It considers side-exits as potential locations for trace header when the execution of a trace is repetitively aborted due to a guard failure. In this case the VM starts recording a new trace from the point where the trace is aborted. Moreover, it generates special nested trace trees for nested loops. On the same path, Chang et. al [31] released a tracing JIT called Tamarin-Tracing in 2009. Tamarin is the Adobe's VM that implements ActionScript 3 [32], a flavour of ECMAScript [33]. JavaScript is the most known flavour of ECMAScript, but most of JavaScript can be executed without modification on Tamarin. Tamarin-Tracing is a branch of Tamarin with a trace-based just-in-time compiler that uses run-time profiling to identify frequently executed code paths. Both TraceMonkey and Tamarin-Tracing were developed with the support of a joint collaboration of Mozilla and Adobe. Others relevant works related to the ones just mentioned are: Gal's PhD thesis [34] in 2006; Gal et al. [22, 35] respectively in 2006, 2007; Chang et.al [36, 37] respectively in 2007, 2011.

A further project has been realised in the context of *meta-tracing* where the JIT compiler does not trace the user program being run, but it traces the execution of the interpreter while it runs this program. In 2009, Bolz et al. [17] applied this technique for PyPy's tracing JIT compiler to programs that are interpreted for dynamic languages, including Python. Many studies had been conducted on the same direction including Bolz et. al [38, 39, 40]

respectively in 2010, 2013, 2014; Bolz's PhD thesis [41] in 2012 ; Cuni's PhD thesis [7] in 2010; Ardö et al. [42] in 2012; Vandercammen's MSc thesis [43] in 2015. On the same path Bauman et al. [44] created Pycket, a tracing JIT for Racket that is a dynamically typed functional programming language descended from Scheme. Pycket is implemented using the RPython meta-tracing framework, which automatically generates a tracing JIT compiler from an interpreter written in RPython (a subset of Python "Restricted Python").

Another important contribution for trace-based just-in-time compilation has been done by Bebenita et al. [45] in 2010. They designed and implemented SPUR, a tracing JIT for Microsoft's CIL (the target language of C#, VisualBasic, F#, and many other languages).

A very successful tracing just-in-time compiler for the Lua programming language is LuaJIT by Mike Pall [3]. Its first version, LuaJIT 1, was released in 2005 provided with a JIT implemented in the assembly language DynASM [5, 46]. In LuaJIT 2, published in 2012, the whole VM has been rewritten from scratch in DynASM realising a fast interpreter and the JIT was reimplemented in C. There is no documentation of the LuaJIT internals, but a short summary of techniques used is given by Pall in a public statement about the intellectual property contained in LuaJIT [47]. In the following years many JITs have been implemented from LuaJIT, because of its outstanding performance as just-in-time compiler. Schilling developed a trace compiler for Haskell based on LuaJIT called Lambdamachine [10] for his PhD thesis in 2013. Another just-in-time compiler that is born from LuaJIT is RaptorJIT [20] by Gorrie. It is a fork of LuaJIT suitable for high-performance low-level system programming. It aims to ubiquitous tracing and profiling to make application performance and compiler behaviour transparent to programmers. It is provided with an interactive tool for inspecting and cross-referencing trace and profiler data called Studio [48]. Finally, another relevant software that should be mentioned in this context is OpenResty [49]. It is a full-fledged web platform that integrates a modified version of LuaJIT.

Chapter 3

Virtual Machine

This chapter introduces a detailed description of the Virtual Machine (VM) presenting the working principles of its components.

3.1 Frontend

The compiler frontend is composed by Lexer, Parser and BC Frontend.

3.1.1 Lexer

The lexer (implemented in `lj_lex.[c,h]`) converts a sequence of Lua intructions into a sequence of tokens. Its input is a Lua program (`*.lua`). It uses *LexState* as a principal data structure. The user-provided *rfunc* function is used to read a chunk of data to process. It is accessed through the *p* and *pe* pointers. The main function is *lex_scan* that dispatches the work to other functions depending on the type of data to be processed (comment, string literal, long string, numbers etc...). TValues (*tokval*, *lookaheadval*) are used to store the token values were *LexToken* (*tok*, *lookahead*) determine the type. The string buffer (*sb*) is used to accumulate characters of a future string before internalizing it. All Lua keyword are internalized as a string at the very beginning, GCstr has the field *reserved* for marking them.

3.1.2 Parser

The parser (implemented in `lj_parse.[c,h]`) takes input in the form of a sequence of tokens produced by the lexer. LuaJIT does not build an abstract syntax tree representation of the parsed code (as a "standard" compiler would build), but it directly generates the bytecode on the fly using helpers from

lj_bc.h. It also uses *LexState* as the principal data-structure. The *lj_parse* function is the entry point and parses the main chunk as a vararg function. The unit of emission is the function *GCproto* and the structure used for the construction is *FuncState*. Parsing is a succession of chunks (*parse_chunk*) were *parse_stmt* is the principal function called for each line, dispatching the work depending on the current token type. *FuncScope* is a linked-list of structure used for scope management.

3.1.3 Bytecode frontend

Another feature of LuaJIT consists of saving and loading the bytecode directly (object files `*.o`), avoiding the chain lexer and parser.

The writing part is handled by the module *bcsave.lua* that use the *lj_bcwrite* function from *lj_bcwrite.c* to generate the data to be written.

The reading part is done by the code in *lj_bcread.c* file. When it is detected that the input file is a bc dump instead of a plain Lua code, *cpparser* from *lj_load.c* calls *lj_bcread* instead of *lj_parse* normally. This reader also use *LexState* as the principal data-structure.

3.2 Internals

3.2.1 Tagged value

LuaJIT represent all internal elements as 64-bits *TValue* (tagged value). It uses the nan-tagging technique to differentiate between numbers and other types of element. In fact, *lua_number* are 64-bits floating-point numbers following the *ieee* standard. In this way, numeric NaNs are canonised by the CPU (`0xfff8` in msb and zero otherwise), letting the possibility to use the lower bits to represent arbitrary data. Internally, LuaJIT has two different representations, one for 32-bits and another for 64-bits (*LJ_GC64*) mode (see Tables 3.1 and 3.2). In those tables, *itypes* are numbers identifying the type of the object. GC objects (Garbage-Collected Object) represent all allocated objects that are managed by the garbage collector. *GCRef* are references to such object.

Table 3.1: Internal object tagging for 32-bits mode

	MSW	**LSW**
size	**32-bits**	**32-bits**
primitive types	itypes	
lightuserdata (32-bits)	itypes	void *
lightuserdata (64-bits)	`0xffff`	void *
GC objects	itypes	GCRef
int	itypes	int
number	double	

Table 3.2: Internal object tagging for 64-bits mode

	MSW			**LSW**
size	**13-bits**	**4-bits**	**15-bits**	**32-bits**
primitive types	1...1	itype	1...1	
lightuserdata	1...1	itype	void *	
GC objects	1...1	itype	GCRef	
int	1...1	itype	0...0	int
number	double			

3.2.2 String Internalization

All strings manipulated by LuaJIT are internalized. This includes strings literals of the user-side Lua code, identifiers and tokens of the Lua language itself, and strings used internally by LuaJIT. With the internalisation mechanism, only one copy of a specific string is kept in memory. If multiple copies of the same string are requested, a pointer to the internalised version of the string is returned (instead of allocating a new string). Strings need to be immutable and are null-terminated. String's function are implemented in the *lj_str.c* file and internalisation is done by the *lj_str_new* function.

LuaJIT implements a hash table and uses a very sparse hash function. Collisions are handled by the use of a singly-chained linked list. The table is resized and all string rehashed when a 100% load is reached. The necessary states are saved in the *global_State* structure in the *lj_obj.h* file.

```
typedef struct global_State {
  GCRef *strhash; /* String hash table (hash chain anchors). */
  MSize strmask;  /* String hash mask (size of hash table - 1). */
  MSize strnum;   /* Number of strings in hash table. */
  [...]
}
```

3.2.3 Lua table

Tables are garbage-collected objects represented by the structure *GCtab* in `lj_obj.h`. The functions to manipulate them are defined in `lj_tab.c` and `lj_tab.h`. *GCtab* is composed of an array part and a hash part. If the array part is small, it is allocated directly after the structure in memory (collocation functionality), otherwise, it is separated. The hash part is a hash table used to store all non-integer key (or integer too big to fit in the array part). It is implemented as an array using a singly-linked list for collision, where nodes of the linked list are within the array (not allocated) and a variation of *Brent's hashing methods* is used. New integer keys that are bigger than the array part are always inserted in the hash part until this one is full. Then, it triggers the resizing of the table. The new *asize* and *hmask* are both power of 2. The new *asize* value corresponds to the biggest power of 2 such that at least 50% of the integers below it are used as keys. The new *hmask* is picked such that all non-integer keys plus the integer keys that are bigger than the new *asize* fit in it. When resizing occur the hash values are re-hashed and integer kes that does fit in the array part is reintroduced there.

The *nomm* field of GCtab is a negative cache for fast metamethods checks. It is a bitmap marking absent fields of the metatable.

3.2.4 Garbage collector

LuaJIT garbage collector (GC) has a tricolour, incremental mark and sweep type implementation. It is presented in the tricolour section of the wiki [50]. Its source code is implemented in `lj_gc.h` and `lj_gc.c`. It uses the *GCState* has a principal structure. The *gc_onestep* function implements the states machine of the GC. The main states with their meaning are shown in the enum below.

```
enum {
  GCSpause,
  GCSpropagate,    /* One gray object is processed. */
  GCSatomic,       /* Atomic transition from mark to sweep phase. */
  GCSsweepstring, /* Sweep one chain of strings from the table. */
  GCSsweep,        /* Sweep a few object from root. */
  GCSfinalize      /* Finalize one userdata or cdata object. */
};

typedef struct GCState {
  GCSize total;          /* Memory currently allocated. */
  GCSize threshold;      /* Memory threshold. */
  uint8_t currentwhite; /* Current white color. */
  uint8_t state;         /* GC state. */
  uint8_t nocdatafin;    /* No cdata finalizer called. */
  uint8_t unused2;
  MSize sweepstr;        /* Sweep position in string table. */
  GCRef root;            /* List of all collectable objects. */
```

```
  MRef sweep;           /* Sweep position in root list. */
  GCRef gray;           /* List of gray objects. */
  GCRef grayagain;      /* List of objects for atomic traversal. */
  GCRef weak;           /* List of weak tables (to be cleared). */
  GCRef mmudata;        /* List of userdata (to be finalized). */
  GCSize debt;          /* Debt (how much GC is behind schedule). */
  GCSize estimate;      /* Estimate of memory actually in use. */
  MSize stepmul;        /* Incremental GC step granularity. */
  MSize pause;          /* Pause between successive GC cycles. */
} GCState;
```

3.2.5 Allocator

LuaJIT has its own embedded allocator which is a customized version of *dlmalloc* (Doug Lea's Malloc). Information on the original implementation can be found on the web article [51] or in the comment of the code [52]. The allocator is implemented in `lj_alloc.c`. Its main structure is *malloc_state*. Memory on the heap is allocated in chunks. Free chunks are managed as double linked-list with the size of the chunk at the beginning and end of it. Unallocated memory is grouped in bins of the corresponding size. There are two types of bins. The smaller one contains chunks of the same size and the top is anchored in *smallbins*. The bigger ones are stored as bitwise digital trees (aka tries) keyed by size where the top of a tree is anchored in *treebins*. The allocator differentiates with two types of memory allocation, if it is higher than 128KB then it asks the operating system for a new memory segment using mmap, if it is lower than that it uses chunks from the current segment. All allocated segments are kept in a linked list anchored in *seg*. For such smaller allocation, the allocator first tries to find an exact fit from the available chunks to optimize for internal fragmentation. If it cannot find one and that the requested size is smaller than *dvsize* then it uses the *dv* chunk (designated victim) which is the last chunk that has been split. This is done to optimise locality. Otherwise, it goes for a best-fit match. If no chunk big enough is available, it asks the system to extend the segment and use the boundary chunk *top* (always kept free). When memory is freed, it does chunk coalescing to avoid memory fragmentation. If *topsize* is bigger than *trim_check*, then the current segment is shrunk and the memory is given back to the OS. *release_checks* is a decreasing counter that when it riches zero triggers a check of all segments to release empty ones back to the OS.

3.2.6 Function

There are two different representations of function, the function's prototype, and the function's closure. Lua function's prototypes are represented by *GCproto* (`lj_obj.h`), and are followed by the functions' bytecodes in memory.

The closures are represented by the *GCfuncL* for Lua function and *GCfuncC* for C function (using Lua API). They contain the necessary information for upvalues. Upvalues are represented by the *GCupval* which contains the corresponding value or a reference to the stack slot with the appropriate value. Closures can be managed by the functions present in `lj_func.c` allowing to create them, destroy them and closing their upvalues.

3.2.7 Fast Function

Fast functions are specially optimized standard library function. There are two parts for each function: implementation and fallback handler. The implementation part, called "the fast pass" handles the general cases. It is realised in assembly in the `vm_*.dasc` files. If the fast pass fails, the VM calls the corresponding fallback handler (all *LJLIB_ASM* marked functions) that will try to recover from the failure, if possible. Examples of recoverable cases might be: wrong argument type if coercion succeeds or stack overflow if stack reallocation succeeds (see comment in `lj_lib.h`).

3.2.8 GC64 mode

By default, LuaJIT use 32-bit pointers which limit its memory on x86 to 4 GB on a 32-bit platform and even 2 GB on a 64-bit platform (due to constant pointers being embedded as offsets in x86 addressing modes, which are signed 32-bit integers). This limitation has recently been overcome with the *GC64* mode (currently in beta version). It allows to fully use the well-known 47-bit address space. To activate this mode you need to uncomment in `src/Makefile` the *LUAJIT_ENABLE_GC64* variable and to perform a full recompilation of LuaJIT.

3.3 Bytecode interpreter

Most of the VM mechanism is written using the DynASM syntax. To get a better understanding of the underlining technology see Appendix B.

The low-level VM code written for each architecture using DynASM is implemented in `vm_(target).dasc` files. It is mainly composed of a huge switch case that generates the assembly code for each bytecode. It also contains code for parts that require low-level implementation, e.g. the stack unwinding, the implementation of the fast path of ASM fast library functions, etc.

3.4 Library

3.4.1 Standard library

LuaJIT supports full compatibility with Lua 5.1, hence it implements the standard library. The code is copied and adapted from the *PUC-RIO* Lua interpreter. A list of the corresponding files and descriptions is shown below.

Table 3.3: Files of the Lua standard library

File name	Description
lib_base.c	Base and coroutine library.
lib_debug.c	Debug library.
lib_init.c	Load and initialize standard libraries.
lib_io.c	Files and I/O library.
lib_math.c	Math library (abs, sqrt, log, random, etc...).
lib_os.c	OS library (date, time, execute, remove, etc...).
lib_package.c	Package library (load, require, etc...).
lib_string.c	String library (gsub, match, etc...).
lib_table.c	Table library (new, clear, insert, foreach, etc...).

3.4.2 LuaJIT extensions

In addition to the standard library, LuaJIT is equipped with some library extensions [53]. Along with few improvements of existing modules, it provides three new extension modules: (i) `lib_bit.c` for bitwise operations [54], (ii) `lib_ffi.c` which contains the implementation of the functions that interact with the FFI library (see Sec. 3.5 for details), (iii) `lib_jit.c` which provides functions to control the behaviour of the JIT compiler engine (see Chapt. 4 for details).

3.4.3 The C API

All those libraries are implemented using the Lua C API that allows to create and manipulate Lua data, manage the Lua stack etc... These functions are implemented in `lj_api.c`, `lj_lib.c` and `lib_aux.c`.

3.4.4 Build Library

LuaJIT uses a subtle technique to automatically generate files that are included during compilation in order to help building and loading the standard

library without the need for manual maintenance. Here are described the different steps and what they are useful for.

Firstly, if no Lua interpreter (either PUC-Lua or LuaJIT) is available on the machine, a simplified and reduced version of Lua interpreter is built from `minilua.c`. Then, the interpreter is used to run `genlibbc.lua` that will be responsible for parsing all LuaJIT's source files searching for the *LJLIB_LUA* macro that surrounds library functions names that are written in Lua. It then generates the `buildbm_libbc.h` file that contains the Lua bytecodes for all those functions in the *libbc_code* array and a mapping of the function name and the bytecodes offset for that function in *libbc_map*.

This newly generated file is built along with all `buildvm_*` files to create the *buildvm* program that is used to parse from the library source code all other *LJLIB_** macro and generates some files (`lj_bcdef.h, lj_libdef.h, lj_ffdef.h, lj_recdef.h and vmdef.lua`) that will be added to LuaJIT compilation. In Table 3.4 it is is shown the description of the macros and in Table 3.5 the description of the corresponding generated file.

Macro	**Description**
LJLIB_MODULE_*	register new module.
LJLIB_CF(name)	register C function.
LJLIB_ASM(name)	register fast function fallback handler.
LJLIB_ASM_(name)	register fast function that uses previous LJLIB_ASM fallback handler.
LJLIB_LUA(name)	register Lua function.
LJLIB_SET(name)	register previous Lua stack value into the module table with *name* has key. • '!' : last stack value became next function's env
LJLIB_PUSH(val)	push *val* on the Lua stask. • 'lastcl' : copy last stack value • 'top-x' : copy last x^{th} stack value • ' "..." ' : push internalized string

LJLIB_REC(handler)	register a handler to record a function. • '.' : get the function's name • 'name data' : – *name* of recorder – auxiliary *data* to put in *recff_idmap*
LJLIB_NOREGUV	to not register this function in module.
LJLIB_NOREG	to not register a function in lj_lib_cf_*.

Table 3.4: Macros used to build the library

File	**Description**
lj_bcdef.h	for each fast functions, *lj_bc_ofs* contains the offset from *lj_vm_asm_begin* (in *lj_vm.h*) to the mcode of the function and *lj_bc_mode* contains the byte code operande mode (all set to *BCMODE_FF*) (see *lj_bc.h* and this *Introduction* section of the wiki [55]).
lj_ffdef.h	list of all library function name
lj_libdef.h	*lj_lib_cf_** arrays contains the list of function pointers for the * library. *lj_lib_init_** are arrays of packed data describing how the corresponding library should be loaded (see *lj_lib_register* in *lj_lib.c* for the function that parse those data).
lj_recdef.h	for each library functions *recff_idmap* contains an optional auxiliary data (opcode, literal) allowing to handle similar functionalities in a common handler. *recff_func* contains the list of record handler.

vmdef.lua	contains all vm definition for use in Lua. • bcnames : bytecode names • irnames : IR instructions names • irfpm : floating point math function names • irfield : maps field id to field name (specific field that the IR instruction is accessing) • ircall : name of special function for calls instruction. • traceerr : maps error num to error message • ffnames : maps library function id to function name

Table 3.5: Generated files description

3.5 Foreign function interface

The FFI (Foreign Function Interface) is one of LuaJIT extensions modules. It allows to call external C functions and use C data structures from pure Lua code. A very small (not representative) example is shown below.

```
1 local ffi = require("ffi")
2 ffi.cdef[[
3   typedef struct points { int x,y,z; } points;
4 ]]
5 local p1 = ffi.new("points", { 1, 2, 3 })
6 print(p1.x) --  1
7 p1.x = 11
8 print(p1.x) -- 11
```

There is some official documentation for FFI users on the official LuaJIT website where you can find the motivation for the FFI module [56], a small tutorial [57], the API documentation [58] and the FFI semantics [59]. There is also a reflection library for FFI ctypes [60] and its documentation [61] for anyone interested in exploring the ctype of a given cdata.

This section will present the internal implementation of the FFI and not its use. The organisation of the information will follow the actual implementation files.

lib_ffi.c

It is the top-level file of the FFI library. It contains the implementation of the FFI API, the function that makes the connection between Lua and C using the standard Lua *C API*. It is also responsible for loading the FFI module (*luaopen_ffi*). This file mainly uses and connects together functionalities implemented in other files. For instance, it is responsible for the allocation and initialisation of the main state (*CTState*) explained below.

lj_obj.h

From this file, we are only interested in this section by the *GCcdata* structure that is the garbage-collected object representing any C data use through and with the FFI. Its structure is shown below. The key point to highlight is the *ctypeid* which is the index of the ctype describing the attached data (the payload follows the structure in memory).

```
typedef struct GCcdata {
  GCHeader;
  uint16_t ctypeid; /* C type ID. */
} GCcdata;
```

lj_ctype.h

The *CType* data structure responsible for describing to the FFI what kind of data the *cdata* represents (e.g. variable, struct, function, etc.). A detailed schema in this regards is shown in Tab. 3.6. The abbreviations used are explained in Tab. 3.7.

```
typedef struct CType {
  CTInfo info;   /* Type info. */
  CTSize size;   /* Type size or other info. */
  CTypeID1 sib;  /* Sibling element. */
  CTypeID1 next; /* Next element in hash chain. */
  GCRef name;    /* Element name (GCstr). */
} CType;
```

The most important struct of the FFI is *CTState*. It contains all the internalize ctype in the *tab* table. *finalizer* is a weak keyed Lua table (values can be garbage collected if the key is not referenced elsewhere) containing all finalizer registered with the *ffi.gc* method. *miscmap* is a Lua table mapping all metatable of ctypes registered using the *ffi.metatype* method in the negative CTypeID range and all callback functions in the positive callback slot range. Any metatable added to miscmap is definitive and never collected. *hash* is an array used as a hash table for quick CTypeID checks. It maps both, the hashed name of named elements and the hashed type (info and

Table 3.6: Summary of CType informations

	info				size	sid	next	name
	type	flags	A	cid				
size	4	8	4	16	32	16	16	GCRef
NUM	0000	BFcvUL..	A		size		type	
STRUCT	0001	..cvu..V	A		size	field	name	name
PTR	0010	..cvR...	A	cid	size		type	
ARRAY	0011	V^2Ccv...V	A	cid	size		type	
VOID	0100	..cv....	A		size		type	
ENUM	0101	.....	A	cid	size	const	name	name
FUNC	0110	....V^3S..	..cc	cid	nargs	field	name	name
TYPEDEF	0111	.....	....	cid			name	name
ATTRIB	1000	.... | attrnum		cid	attr	sib	type	
FIELD	1001	.....	....	cid	offset	field		name
BITFIELD	1010	B.cvU | csz		.bsz | .pos	offset	field		name
CONSTVAL	1011	..c.....	....	cid	value	const	name	name
EXTERN	1100	.....	....	cid		sib	name	name
KW	1101	.....	....	tok	size		name	name

Table 3.7: Definition of CType fields

flags		***A***	
B	Boolean	A	allignement of 2^A bytes
F	Float	cc	calling convention
c	const	attrnum	Attributes number (see CTA_* in lj_ctype.h)
v	volatile	csz	size of the memory slot
U	Unsigned	***cid***	
L	Long	cid	child id
u	union	bsz	number of bits of the bit-field
V	VLA	pos	starting position in bit inside the memory slot
R	reference	tok	token id
V^2	Vector	***size***	
C	Complex	size	size in bytes
V^3	Vararg	nargs	number of fixed arguments of a function
S	SSE arguments	offset	offset in bytes from the start of the struct
		value	the actual constant value
		attr	value of the attribute
		sib	
		field	function args or struc/union field
		const	constant value
		sib	chain of attribute or object of the attribute
		name	GCRef to a string containing the identifier.
		next	chain for hash collisions in cts->hash

size) for unnamed elements to the corresponding CTypeID. Collisions are handled in a linked list using the *next* field of the *CType* struct.

```
typedef struct CTState {
  CType *tab;         /* C type table. */
  CTypeID top;        /* Current top of C type table. */
  MSize sizetab;      /* Size of C type table. */
  lua_State *L;       /* Lua state (for errors and allocations). */
  global_State *g;    /* Global state. */
  GCtab *finalizer;   /* Map of cdata to finalizer. */
  GCtab *miscmap;     /* Map -CTypeID->metatable and cb slot->func. */
  CCallback cb;       /* Temporary callback state. */
  CTypeID1 hash[...];/* Hash anchors for C type table. */
} CTState;
```

lj_ctype.c

This file contains functions to manage *CType*. It is divided into three parts. The first one for the allocation, creation and internalization of *CType*. The second one provides getters to retrieve C type information. The last one is type representation, providing the necessary functions to convert a *CType* to a human-readable string representation. *lj_ctype_repr* is the entry function that returns the internalize string representation. It uses the struct *CTRepr* to create the representation by appending/prepending characters through the pb/pe pointers into the buffer. The main function is the *ctype_repr* that contains a switch on the *CType* info.

```
typedef struct CTRepr {
  char *pb, *pe; /* Points to begining/end inside the buffer*/
  CTState *cts;  /* C type state. */
  lua_State *L;
  int needsp;    /* Next append needs an extra space character */
  int ok;        /* Indicate if buf is currently a valid type */
  char buf[...]; /* String buffer of the ctype being constructed */
} CTRepr;
```

lj_cparse.h

The *cparser* is responsible for parsing the string of the C declarations identifying types or external symbols. Its code structure is quite close to the Lua lexer/parser. Its principal struct is the *CPState* which is similar to *LexState*. In this struct, *tmask* is a mask constraining the possible ctype of the next identifier. The *mode* defines the behaviour of the parser with respect to the input. It has a different behaviour according to the type: accepting multiple declarations, skipping errors, accept/reject abstract declarators, accept/reject implicit declarators etc... (see CPARSE_MODE_* for a full definition).

```
typedef struct CPState {
  CPChar c;               /* Current character. */
  CPToken tok;            /* Current token. */
```

```
  CPValue val;            /* Token value. */
  GCstr *str;             /* Interned string of identifier/keyword. */
  CType *ct;              /* C type table entry. */
  const char *p;          /* Current position in input buffer. */
  SBuf sb;                /* String buffer for tokens. */
  lua_State *L;           /* Lua state. */
  CTState *cts;           /* C type state. */
  TValue *param;          /* C type parameters. ($xyz)*/
  const char *srcname;    /* Current source name. */
  BCLine linenumber;      /* Input line counter. */
  int depth;              /* Recursive declaration depth. */
  uint32_t tmask;         /* Type mask for next identifier. */
  uint32_t mode;          /* C parser mode. */
  uint8_t packstack[...]; /* Stack for pack pragmas. */
  uint8_t curpack;        /* Current position in pack pragma stack. */
} CPState;
```

lj_cparse.c

This file contains the code of a simple lexer and a simplified, not valid C parser. It uses the *CPState* for the parsing of the input string and the *CPDecl* structure for the construction of the corresponding CType. During parsing the chain of typdef is unrolled (typdef are still internalized for future reference but are not chained to the created ctype).

```
typedef struct CPDecl {
  CPDeclIdx top;      /* Top of declaration stack. */
  CPDeclIdx pos;      /* Insertion position in declaration chain. */
  CPDeclIdx specpos; /* Saved position for declaration specifier. */
  uint32_t mode;      /* Declarator mode (same as CPState) */
  CPState *cp;        /* C parser state. */
  GCstr *name;        /* Name of declared identifier (if direct). */
  GCstr *redir;       /* Redirected symbol name. */
  CTypeID nameid;     /* Existing typedef for declared identifier. */
  CTInfo attr;        /* Attributes. */
  CTInfo fattr;       /* Function attributes. */
  CTInfo specattr;    /* Saved attributes. */
  CTInfo specfattr;  /* Saved function attributes. */
  CTSize bits;        /* Field size in bits (see Ctype bsz). */
  CType stack[...];  /* Type declaration stack. */
} CPDecl;
```

lj_cdata.c

This file contains the functions responsible for cdata management, such as allocations, free, finaliser, getter, setter and indexing.

lj_cconv.c

This file is responsible for ctype conversion. It is divided in 5 parts: (i) C type compatibility checks, (ii) C type to C type conversion, (iii) C type to TValue conversion (from C to Lua : i.e returned values), (iv) TValue to C

type conversion (from Lua to c: i.e passed arguments), and (v) initializing C type with TValues (Initialization of struct/union/array with Lua object).

lj_carith.c

This file contains the implementation for all built-in cdata arithmetic, such as pointers arithmetic and integer arithmetic. It mainly manipulates some *CDArith* structure shown below.

```
typedef struct CDArith {
  uint8_t *p[2]; /* data  of the two operands */
  CType  *ct[2]; /* ctype of the two operands */
} CDArith;
```

lj_ccall.c

This file contains the code handling calls to C function. It realises struct/array register classification (see CCALL_RCL_*) computing how it can be passed as argument/return values (in GP register, SSE register or memory). It handles then the decomposition/packing depending on the calling convention picked.

lj_ccall.h

This file contains the main structure used for C function call.

```
typedef struct CCallState {
  void (*func)(void); /* Pointer to called function. */
  uint32_t spadj;     /* Stack pointer adjustment. */
  uint8_t nsp;        /* Number of stack slots. */
  uint8_t retref;     /* Return value by reference. */
  uint8_t ngpr;       /* Number of arguments in GPRs. */
  uint8_t nfpr;       /* Number of arguments in FPRs. */
  [...]
  FPRArg fpr[...];    /* Arguments/results in FPRs. (SSE) */
  GPRArg gpr[...];    /* Arguments/results in GPRs. */
  GPRArg stack[...];  /* Stack slots. */
} CCallState;
```

lj_ccallback.c

This file provides the FFI C callback handling. The principal structure is the *CCallback* (see `lj_ctype.h`) that mainly use through the *cb* field of the *CTState* structure. Each callback is associated with a unique *cb slot* and *cts->miscmap* contains the mapping between *cb slot* and function pointers. The *cts.cb.cbid* is a table mapping *cb slot* to the corresponding *CTypeID*. *cts.cb.mcode* is an mapped executable page that contains a push of slot id and a jump to *lj_vm_ffi_callback* (the entry in this page corresponds to the *cb*

address provided to the C code). The order of call when a callback occurs is the following:

- C calls a callback address
- It arrives in the callback mcode page (push the appropriate slot id and call *lj_vm_ffi_callback*)
- It arrives in *lj_vm_ffi_callback* that charge in registers the callback.fpr/gpr + stack in registers
- It arrives in *lj_ccallback_enter* that prepare the Lua state and do conversion of argument from C to Lua type.
- It arrives in *lj_vm_ffi_callback*: execute the callback with this Lua state
- Lua callback
- It arrives in $|->$*cont_ffi_callback*: call lj_ccallback_leave
- It arrives in *lj_ccallback_leave* convert return value from Lua to C.
- It arrives in $|->$*cont_ffi_callback*: return to C code with the return value in register/stack.
- It arrives in C code.

```
typedef struct CCallback {
  FPRCBArg fpr[...]; /* Arguments/results in FPRs. */
  intptr_t gpr[...]; /* Arguments/results in GPRs. */
  intptr_t *stack;   /* Pointer to arguments on stack. */
  void *mcode;       /* Machine code for callback func. pointers. */
  CTypeID1 *cbid;    /* Callback type table. */
  MSize sizeid;      /* Size of callback type table. */
  MSize topid;       /* Highest unused callback type table slot. */
  MSize slot;        /* Current callback slot. */
} CCallback;
```

lj_clib.[c,h]

This file contains the necessary code to load/unload ffi library. It also handles the indexing of the external library using named symbol. It uses platform-specific tools to explore the exposed symbols. Every symbol is resolved only once and cached in the *CLibrary* cache table.

```
typedef struct CLibrary {
  void *handle; /* Opaque handle for dynamic library loader. */
  GCtab *cache; /* Cache resolved symbols. Anchored in ud->env. */
} CLibrary;
```

Chapter 4

JIT compiler

In this chapter it will be illustrates how LuaJIT performs trace-base just-in-time compilation. We will investigate how the JIT compiler create a trace and all the aspect related to it.

Traces generation in LuaJIT follows the canonical working principles characteristic of tracing JITs (see Chapter 2):

- LuaJIT begins executing bytecode instructions with the interpreter (VM).
- While executing bytecode instructions, the VM monitors the execution frequency of potential candidates for hotpath headers (hotloops/hotfunctions).
- When a hotpath header is identified, LuaJIT starts recording. It executes bytecode with the VM and it records each executed instruction in parallel.
- The IR is incrementally generated in SSA (Static Single Assignment) form.
- The IR is optimised applying traditional compiler optimisation.
- After recording and optimisation, LuaJIT compiles the trace emitting mcode specific to the architecture.
- The bytecode is patched with special bytecode instructions (`J...`) that force the execution of JIT-compiled loops/functions instead of interpreting them.
- Any stage of recording, optimisation or compilation might raise an error that would abort the trace creation.

- When the same hotcode generate too many times a trace abort, it is blacklisted. LuaJIT will never try to record it again. The bytecode of that hotloop/hotfunction is patched with a special bytecode instruction (`I...`) that stops hotspot detection and force execution in the interpreter.

4.1 Hotpaths detection

The mechanism used by LuaJIT to detect hotpaths is based on Natural Loop First (NLF) and counter base profiling (see Sec. 2.3.1 2.3.2). Both loops and functions can generate traces, but hotloops are preferred over hotfunctions.

Trace heads selection is accomplished with a well-defined implementation. Some bytecode instructions are considered as "special" because they are the only ones that can lead to potential hotpath headers (see table 4.1).

Operation	Description
FORL	Numeric 'for' loop
LOOP	Generic loop
ITERL	Iterator 'for' loop
ITERN	Specialized iterator function next() (NYI)
FUNCF	Fixed-arg Lua function
FUNCV	Vararg Lua function

Table 4.1: Bytecode instructions for hotpaths detection

In order to count the invocation frequency of these instructions, LuaJIT uses a relatively small (64 entries) hash table, containing 16-bit integer counters.

```
typedef uint16_t HotCount;

/* Number of hot counter hash table entries (must be a power of two). */
#define HOTCOUNT_SIZE   64

/* Global state */
typedef struct GG_State {
  ...
  HotCount hotcount[HOTCOUNT_SIZE]; /* Hot counters. */
  ...
} GG_State;
```

Listing 4.1: `lj_dispatch.h`

Index	value
0	111
1	108
2	111
3	51
...	...
63	111

Table 4.2: Example snapshot of an Hotcount Table

Each counter is initialised by default to 111 and it is decremented by 2 when tracing hotloops or by 1 when tracing hotfunctions. This is how LuaJIT gives preference to natural loops.

```
/* Hotcount decrements. */
#define HOTCOUNT_LOOP   2
#define HOTCOUNT_CALL   1

#define JIT_PARAMDEF(_) \
  _(\007, hotloop,  56) /* # of iter. to detect a hotloop/call. */ \

/* Initialize hotcount table. */
void lj_dispatch_init_hotcount(global_State *g)
{
  int32_t hotloop = G2J(g)->param[JIT_P_hotloop]; /* extract hotloop value = 56 */
  HotCount start = (HotCount)(hotloop*HOTCOUNT_LOOP - 1); /* start = 111 */
  HotCount *hotcount = G2GG(g)->hotcount;
  uint32_t i;
  for (i = 0; i < HOTCOUNT_SIZE; i++)
    hotcount[i] = start; /* init hotcouters to 111 */
}
```

Listing 4.2: `lj_dispatch.[c,h], lj_jit.h`

Every time a "special" instruction is executed the corresponding counter in the hash table is decremented. When the counter reaches zero, the VM starts recording.

The hash function used is a *modulus* 64 operation of the program counter (PC), which is a true pointer in the process memory. The PC contains the virtual memory address of the current bytecode instruction. Thus, when executing a "special" bytecode instruction the VM decrements a counter at the index `(PC/4)%64` in the hotcount table. Consecutive bytecode instructions are stored at consecutive memory location with a step of 4 bytes. For 32 bits architecture the PC has a shape such as `0x419628d0`, while for 64 bits architecture it is `0x7fd7493e56dc`.

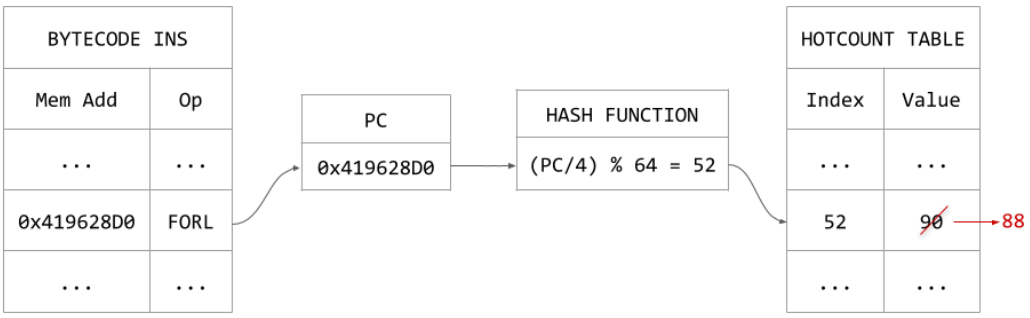

Figure 4.1: Hotcount decrement for loops

Indeed, to set and get hotcount values the following macros are defined.

```
#define hotcount_get(gg, pc) \
  (gg)->hotcount[(u32ptr(pc)>>2) & (HOTCOUNT_SIZE-1)] /* hotcount[(PC/4)%64] */
#define hotcount_set(gg, pc, val) \
  (hotcount_get((gg), (pc)) = (HotCount)(val))
```

Listing 4.3: `lj_dispatch.h`

4.1.1 Architecture specific implementation

The hotpath detection technique just explained is fully implemented in assembly (in the `vm_ARCH.dasc` architecture specific files).

To give an example, `vm_x86.dasc` contains a macro (`hotloop`) that is executed when a "special" loop instruction occurs (i.e. `FORL`). In `hotloop`, the counter is decremented by 2 and, when it reaches zero, the execution flow jumps to `->vm_hotloop`. Finally, the VM calls an external C function `lj_trace_hot` (defined in `lj_trace.c`) where the counter is reset to 112 and LuaJIT starts recording.

```
/* Generate the code for a single instruction. */
static void build_ins(...) {
  ...
  switch (op) {
    ...
    case BC_FORL:
    |.if JIT
    |  hotloop RB
    |.endif
    break;
    ...
  }
}

|// Decrement hashed hotcount and trigger trace recorder if zero.
|.macro hotloop, reg
|  mov reg, PC
|  shr reg, 1
|  and reg, HOTCOUNT_PCMASK
```

```
|  sub word [DISPATCH+reg+GG_DISP2HOT], HOTCOUNT_LOOP
|  jb ->vm_hotloop
|.endmacro

|// hotloop counter underflow.
|->vm_hotloop:
|.if JIT
|  ...
|  call extern lj_trace_hot@8 // (jit_State *J, const BCIns *pc)
|  jmp <3
|.endif
```

Listing 4.4: `vm_x86.dasc`

The same logic is applied for hotfunctions. If a "special" function call instruction is executed the VM decrements a counter by 1 in the hotcount table (using the macro `hot_call`). When the counter reaches zero the execution flow jumps to `->vm_hotcall`. In this case the VM calls an external C function `lj_dispatch_call` defined in `lj_dispatch.c`.

The example above refers to x86 architectures, but similar procedures are used for the other architectures.

4.1.2 Hotcount collisions

The hash function selected and the use of a relatively small hotcount table lead inevitably to collisions. When the hash function gives the same result, different "special" bytecode instructions (and therefore different potential hotpaths) correspond to the same counter.

In tracing JITs, as mention by Gal et al. [16], collisions are intentionally tolerated as their impact on the code is relatively limited. Collisions can lead to overestimation of the "hotness" of a code region, triggering "cold" code recording. Thus, false positive may occur. This overestimation may cause slight performance degradation as the compilation cost could be more expensive than simple interpretation of the code. However this degradation can be neglected when analysing the overall performances, especially for very fast JITs such as LuaJIT.

4.1.3 Memory address randomisation

The mechanism used to manage hot counters depends on the memory address where bytecode instructions are located. This technique was adopted because each counter must be attached to the code fragment from where a hotpath is generated. However, this method implies some difficulties in studying the JIT behaviour when the operating systems support Address Space Layout Randomisation (ASLR). ASLR is a memory-protection system that guards

against buffer-overflow attacks by randomising the location where executables are loaded into memory by the operating system. Two consecutive runs of the exact same code will generate different memory address for the same bytecode instruction. Code fragments could be attached to different counters of the Hotcount table and the behaviour of the JIT can be different from run to run.

Most operating systems nowadays support ASLR, but this should not be seen as a problem for LuaJIT. Even if code fragments are attached to different counters in the Hotcount table from run to run, LuaJIT tries to guarantee on average relatively similar performance on the whole application. However, in some cases there are peaks of substantial slowness in execution.

ASLR brought significant difficulties in the context of this research while studying LuaJIT. In fact, the approach adopted to overcome this problem was to disable ASLR when examining LuaJIT internals. In this way the JIT behaviour will be deterministic and bytecode instructions will be stored in the same memory addresses from run to run.

4.2 Recording

Once an hotpath has been triggered LuaJIT starts recording. From the hotpath header, bytecode instructions will be recorded while they are executed. Recording continues until either an end-of-trace condition is encountered or the trace is aborted. The control flow is flattened, therefore only taken branches are recorded and functions are generally inlined.

4.2.1 Trace compiler state machine

The trace compiler (implemented in `lj_trace.c`) manages the recording phases. Its behaviour changes according to its current state (possible states are shown in Tab. 4.3). Each state can be either *active* if the *activation bit* is set to 1 (`0x1n`) or *not Active* if the activation bit is set to 0 (`0x0n`). When an error occurs the trace compiler changes its current state to *not active* before switching to state `LJ_TRACE_ERR` and abort.

```
/* Trace compiler state. */
typedef enum {
  LJ_TRACE_IDLE, /* Trace compiler idle */
  LJ_TRACE_ACTIVE = 0x10,
  LJ_TRACE_RECORD, /* BC recording active */
  LJ_TRACE_START, /* New trace started */
  LJ_TRACE_END, /* End of trace. */
  LJ_TRACE_ASM, /* Assemble trace */
  LJ_TRACE_ERR /* Trace aborted with error */
```

```
} TraceState;
```

Listing 4.5: `lj_jit.h`

Trace compiler state	Active	Not Active
LJ_TRACE_IDLE	0x00	0x00
LJ_TRACE_ACTIVE	0x10	0x00
LJ_TRACE_RECORD	0x11	0x01
LJ_TRACE_START	0x12	0x02
LJ_TRACE_END	0x13	0x03
LJ_TRACE_ASM	0x14	0x04
LJ_TRACE_ERR	0x15	0x05

Table 4.3: Trace compiler state encoding

The trace compiler behaviour is coordinated by a finite state machine (Fig. 4.2) implemented in the function `trace_state` at `lj_trace.c`.

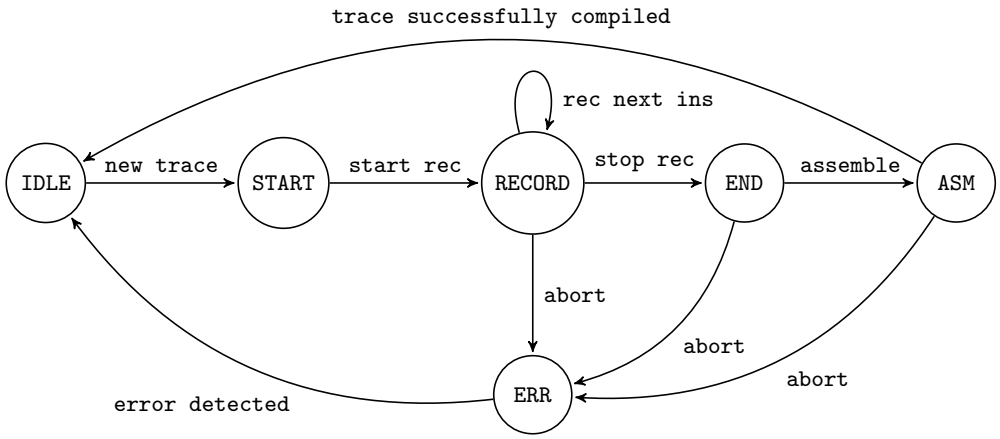

Figure 4.2: Trace compiler state machine

```
/* State machine for the trace compiler*/
static TValue *trace_state(...) {
  do {
  retry:
    switch (J->state) {
    case LJ_TRACE_START:
      J->state = LJ_TRACE_RECORD;
      trace_start(J);
      lj_dispatch_update(J2G(J));
      break;

    case LJ_TRACE_RECORD:
      setvmstate(J2G(J), RECORD);
      lj_vmevent_send_(L, RECORD, ...)
      lj_record_ins(J);
      break;

    case LJ_TRACE_END:
      setvmstate(J2G(J), OPT);
      /* Perform optimisations */
      lj_opt_dce(J);
      /* Loop optimisation failed? */
      if (lj_opt_loop(J)) {
        ...
        /* Try to continue recording*/
        J->state = LJ_TRACE_RECORD;
```

```
      break;
    }
    lj_opt_split(J);
    lj_opt_sink(J);
    J->state = LJ_TRACE_ASM;
    break;

  case LJ_TRACE_ASM:
    setvmstate(J2G(J), ASM);
    lj_asm_trace(J, &J->cur);
    trace_stop(J);
    setvmstate(J2G(J), INTERP);
    J->state = LJ_TRACE_IDLE;
    lj_dispatch_update(J2G(J));
    return NULL;

  default:
    /* Trace aborted asynchronously*/
    setintV(L->top++, LJ_TRERR_RECERR);
    /* fallthrough */

  case LJ_TRACE_ERR:
    if (trace_abort(J)) {
      goto retry;
    }
    setvmstate(J2G(J), INTERP);
    J->state = LJ_TRACE_IDLE;
    lj_dispatch_update(J2G(J));
    return NULL;
  }

 } while (J->state > LJ_TRACE_RECORD);
 return NULL;
}
```

Listing 4.6: `lj_trace.c`

The following paragraphs describe in details each of these states.

4.2.2 Start recording

As mentioned in the previous paragraph, when a hotloop is detected the VM calls an external C function `lj_trace_hot`. For hotfunctions the VM calls `lj_dispatch_call`, but the execution flow also goes to `lj_trace_hot` after some initialisations. Thus, `lj_trace_hot` can be considered as the starting point for trace recording. In this function the counter is reset to 112, the state is changed to `LJ_TRACE_START` and the execution flows goes to `lj_trace_ins`, which begins the trace compiler state machine previously described.

Figure 4.3: Start recording

When the state is `LJ_TRACE_START` the following actions are performed: (i) the trace compiler state is changed to `LJ_TRACE_RECORD`; (ii) the function `trace_start` performs initial setup to start a new trace; (iii) the function `lj_dispatch_update` prepares the dispatcher, so that each bytecode instructions executed by the VM will be henceforward recorded.

4.2.3 Recording

Recording will be done executing in an infinite loop `lj_dispatch_ins` and `lj_trace_ins` until recording stops or an error occurs. The trace compiler state is `LJ_TRACE_RECORD` and for each instruction the execution flow goes to `lj_record_ins` (defined in `lj_record.c`). This function is responsible for recording a bytecode instruction before it is executed. It contains a huge switch case on all possible bytecodes. Finally, from each bytecode instruction, LuaJIT emits the corresponding IR instruction. Therefore, the IR is incrementally generated.

Figure 4.4: Recording a bytecode instruction

If no error occurs, the loop is iterated n times, where n is the number of bytecode instructions of the hotpath.

When the trace compiler records the last bytecode instruction of the hotcode (e.g. `FORL`), the function `lj_record_stop` is executed. It stops recording and sets the trace compiler state to `LJ_TRACE_END`.

4.2.4 Ending recording

Once the recording is concluded, LuaJIT applies optimisations on the IR in SSA form: dead code elimination, loop optimisation, split pass and sink optimisation. It should be noted that most optimisations are performed on-the-fly once all the IR in SSA form is emitted. Hence, eliminated IR instructions are either simply not emitted or ignored during mcode generation [62]. Finally, the trace compiler state is changed to `LJ_TRACE_ASM`.

4.3 Optimisation

It is possible to distinguish between three types of optimisation in the code. First of all, the optimisations of the optimisation engine. They are implemented in the `lj_opt_*.c` files, hence they can be easily identified. These optimisations are can be either: (i) global optimisations, which are run on the entire IR once at the end of the recording phase, during the *LJ_TRACE_END*

state (see 4.3.1, 4.3.2, 4.3.3, 4.3.4) or (ii) local optimisations, which are applied while recording a trace (see 4.3.5, 4.3.6, 4.3.6). Finally, there is a plethora of optimisations and heuristics applied in various parts of the code (ssee LuaJIT wiki on optimisation [63]).

4.3.1 Dead code elimination

Dead Code Elimination (DCE) is performed by the *lj_opt_dce main* function in two phases. During the first phase, which is called "mark snap", it marks all IR instructions that are referenced by a snapshot. In the second phases, which is called "propagate", it iteratively marks all IR instruction that are referenced by an already marked IR instruction while replacing non-marked IR instruction by nops.

4.3.2 Loop optimisations

Loop optimisation is a way to improve code hoisting for traces based on loops. In fact, LuaJIT should try to hoist most invariant instruction and guards in such a way that traces that do not match the current dynamic profile of the code (assumptions on data or type) are exited as soon as possible. Unfortunately, due to the dynamic nature of the IR generated, a trace contains many guards which are control-dependent, making little room for loop-invariant code motion (LICM). The solution used here is a copy-substitution of the body of the loop. It basically consists in always unrolling the loop once before the actual loop instruction. This allows the code executed inside the loop to contains only variant instructions.

4.3.3 Split optimisations

The split optimisation is only used by 32-bits architecture and splits the 64-bits IR instructions into multiple 32-bits once.

4.3.4 Sinking optimisations

This is a very useful optimisation that allows to avoid many temporaries and unnecessary memory accesses and allocations by keeping the values of interest directly in register. Since memory modifications are not performed, we need a way to remember the writes in case those value escape the execution path (they are not temporary anymore). For this purpose snapshots are used (see Section 4.6). This optimisation is implemented in lj_opt_sink.c.and a detailed explanation of it is available on the wiki [64].

4.3.5 Narrowing optimisations

LuaJIT performs the narrowing of Lua numbers (double) into integers when it seems to be useful. It uses demand-driven (by the backend) narrowing for index expressions, integer arguments (FFI), bit operations and predictive narrowing for induction variables. It emits overflow check instruction when necessary. Most arithmetic operations are never narrowed. More details are illustrated in the comment section of `lj_opt_narrow.c`.

4.3.6 Fold engine

The fold engine implement a rule-based mechanism. Rules are declared using the *LJFOLD* macro which contains the IR opcode and a rule on the parameters it applies. During the LuaJIT buildvm (more precisely the `buildvm_fold.c` file), these rules are scanned and the `lj_foldef.h` file is generated. It contains a semi-perfect hash table for constant-time rule lookups, where each entry respect the format depicted in Table 4.4. It also contains a second table with the function to call if a corresponding rule matches.

Table 4.4: Fold hash table, bit pattern entry

8 bits	7 bits	7 bits	2 bits	8 bits
index fold table	fold instr opcode	left instr opcode	00	right instr opcode
index fold table	fold instr opcode	left instr opcode	literal field	

Fold optimisations

The fold optimisations preformed by the FOLD engine are implemented in the `lj_opt_fold.c` file. They can be classified in five well-known techniques: (i) constant folding, (ii) algebraic simplifications, (iii) re-association, (iv) common sub-expression elimination, and (v) array bounds check elimination.

Memory access optimisations

The memory access optimisation perform by the FOLD engine and implemented in `lj_opt_mem.c`. It consists of three components: (i) the alias analysis using high-level semantic disambiguation, (ii) load to load and store to load forwarding, and (iii) finally dead-store elimination.

4.4 Assemble trace

When the compiler state machine reaches the state `LJ_TRACE_ASM`, the trace is assembled. The main function responsible for this task is `lj_asm_trace` (defined in `lj_asm.c`). Each IR instruction previously generated is assembled through the function `asm_ir` that contains a huge switch case on all possible IR codes.

The implementation of the assembler is divided in three different files: (i) `lj_asm.c` contains the platform-independent code, (ii) `lj_asm_ARCH.h` contains the architecture dependent code (e.g. x86), and (iii) `lj_emit_ARCH.h` contains the helper functions to generate instructions for a specific instruction-set. An IR instruction can be translated into $M \geq 1$ machine code instructions.

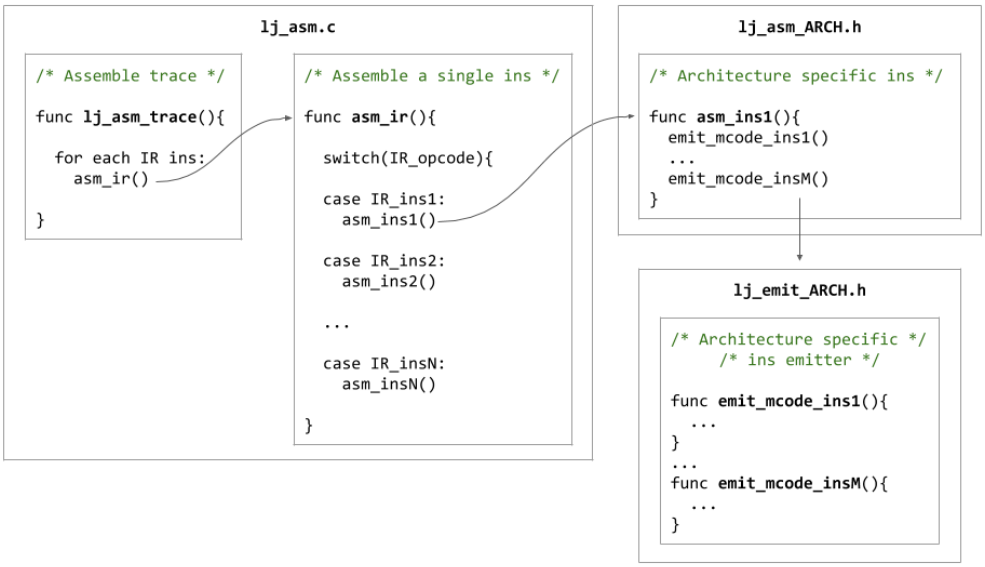

Figure 4.5: Assemble trace

At the end of the assemble phase the hotpath header bytecode is patched with the adequate `J...` operation. This will force the VM to execute the JIT-compiled trace, instead of interpreting the corresponding bytecode instructions. When the execution of the trace will be completed, the control flow goes back to the VM that restart interpreting the bytecode instructions after the hotcode.

The function `trace_stop` is responsible to stop tracing and to patch the bytecode (see Tab. 4.5).

lj_trace.c

```
/* Stop tracing */

func trace_stop(){

  op = bc_opcode(trace_start_ins)

  /* patch bytecode */
  switch(op){

  case BC_ins1:
    patch bytecode with BC_Jins1

  case BC_ins2:
    patch bytecode with BC_Jins1

  case BC_insN:
    patch bytecode with BC_Jins1
  }

  /* Commit new mcode only after all patching is done */
  lj_mcode_commit()

  /* Save current trace */
  lj_trace_save()
}
```

Figure 4.6: Stop tracing

'Standard' op	JIT-compiled op	Description
FORI	JFORI	Numeric 'for' loop init
FORL	JFORL	Numeric 'for' loop
LOOP	JLOOP	Generic loop
ITERL	JITERL	Iterator 'for' loop
FUNCF	JFUNCF	Fixed-arg Lua function
FUNCV	JFUNCV	Vararg Lua function

Table 4.5: Bytecode instructions to force JIT-compiled trace execution

4.5 Trace Abort

4.5.1 Abort

An abort can occur at any stage of the just-in-time compilation: recording, optimisation or assembling. Tab. 4.6 collects all possible causes of abort.

Err Num	Err Code	Description
/* Recording */		
0	RECERR	error thrown or hook called during recording
1	TRACEUV	trace too short

2	TRACEOV	trace too long
3	STACKOV	trace too deep
4	SNAPOV	too many snapshots
5	BLACKL	blacklisted
6	RETRY	retry recording
7	NYIBC	NYI: bytecode %d
/* Recording loop ops */		
8	LLEAVE	leaving loop in root trace
9	LINNER	inner loop in root trace
10	LUNROLL	loop unroll limit reached
/* Recording calls/returns */		
11	BADTYPE	bad argument type
12	CJITOFF	JIT compilation disabled for function
13	CUNROLL	call unroll limit reached
14	DOWNREC	down-recursion, restarting
15	NYIFFU	NYI: unsupported variant of FastFunc %s
16	NYIRETL	NYI: return to lower frame
/* Recording indexed load/store */		
17	STORENN	store with nil or NaN key
18	NOMM	missing metamethod
19	IDXLOOP	looping index lookup
20	NYITMIX	NYI: mixed sparse/dense table
/* Recording C data operations */		
21	NOCACHE	symbol not in cache
22	NYICONV	NYI: unsupported C type conversion
23	NYICALL	NYI: unsupported C function type
/* Optimisations */		
24	GFAIL	guard would always fail
25	PHIOV	too many PHIs
26	TYPEINS	persistent type instability
/* Assembler */		
27	MCODEAL	failed to allocate mcode memory
28	MCODEOV	machine code too long
29	MCODELM	hit mcode limit (retrying)
30	SPILLOV	too many spill slots
31	BADRA	inconsistent register allocation
32	NYIIR	NYI: cannot assemble IR instruction %d
33	NYIPHI	NYI: PHI shuffling too complex
34	NYICOAL	NYI: register coalescing too complex

Table 4.6: Trace compiler error messages

An *asynchronous* trace abort is detected by the trace compiler state machine when the current state does not match any of the possible *active* states. The execution flow ends up in the `default` event of the switch case and recoding aborts.

On the other hand, when a trace aborts *synchronously* the function `lj_trace_err` is called. This throws an error and the current state is set to *not active*. In `lj_trace_ins` the function call of `trace_state` through `lj_vm_cpcall` will return zero, thus the trace compiler state changes to `LJ_TRACE_ERR`. In this case the trace will abort and, if it is a root trace, the PC of the starting bytecode instruction is penalised (penalisation and blacklisting are explained in the following section).

The functions called to throw an error and to abort are shown respectively in Fig. 4.7 and Fig. 4.8

For some abort causes it is quite intuitive to catch what they are supposed to mean (e.g. error thrown, trace too short, long or deep, too many snapshots). For others, it can be more tricky to get their real meaning. Error 5 (blacklisted) means that while recording a trace, the interpreter hits a blacklisted bytecode instruction. In this case recording aborts and the execution goes back to the interpreter. LuaJIT does not allow to retry compilation of blacklisted code fragments in a different context. NYI errors mean Not-Yet-Implemented features of the tracing JIT. All aspects of Lua are implemented in LuaJIT's interpreter, but not all of them are implemented in LuaJIT's JIT compiler. When recording encounters a bytecode instruction which is not-yet-implemented in its corresponding JIT version, trace creation is aborted.

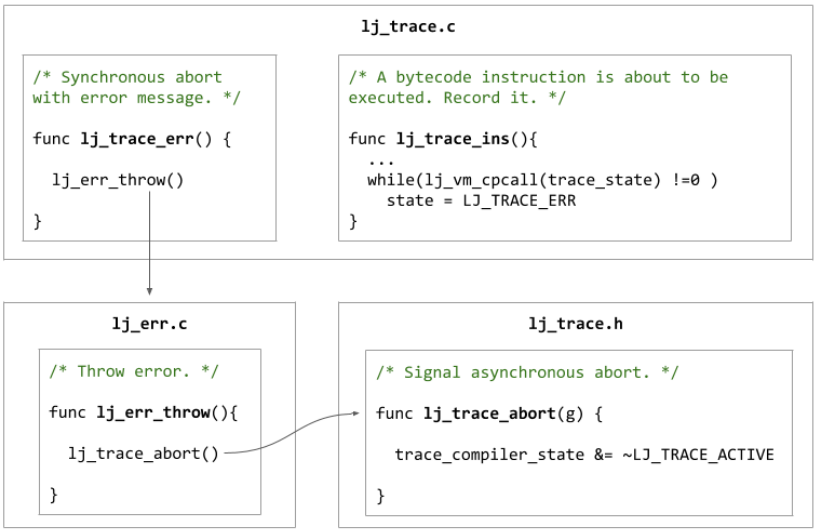

Figure 4.7: Throw error

Figure 4.8: Abort

Many aborts are influenced by the value of some parameters that users can set for the JIT compiler. The most important are shown in Tab. 4.7

Parameter	Deafult	Description
maxtrace	1000	Max. number of traces in the cache
maxrecord	4000	Max. number of recorded IR instructions
maxirconst	500	Max. number of IR constants of a trace
maxside	100	Max. number of side traces of a root trace
maxsnap	500	Max. number of snapshots for a trace
hotloop	56	Number of iterations to detect a hotloop or hot call
hotexit	10	Number of taken exits to start a side trace
tryside	4	Number of attempts to compile a side trace
instunroll	4	Max. unroll factor for instable loops
loopunroll	15	Max. unroll factor for loop ops in side traces
callunroll	3	Max. unroll factor for pseudo-recursive calls
recunroll	2	Min. unroll factor for true recursion
sizemcode	32	Size of each machine code area in KBytes (Windows: 64K)
maxmcode	512	Max. total size of all machine code areas in KBytes

Table 4.7: Parameters of the JIT compiler

4.5.2 Blacklisting

LuaJIT implements blacklisting (see Sec. 2.3.4 for details) in order to prevent recording traces that it tried to generate, but it failed many times.

When hotpath recording fails, the trace compiler increments a counter, so-called *backoff* counter by Gal et al. [19], linked to the hotcode that it tried to record unsuccessfully. Once this counter exceeds a certain threshold, the VM will never try to compile that hotcode again.

To avoid retrying compilation, the bytecode of the hotloop/hotfunction is patched with an operation that stops hotspot detection and force execution in the interpreter. Operations that force interpretation have the same syntax of 'standard' operation with `'I'` as prefix (see table 4.8).

'Standard' op	Force interpreter op	Description
`FORL`	`IFORL`	Numeric 'for' loop
`LOOP`	`ILOOP`	Generic loop
`ITERL`	`IITERL`	Iterator 'for' loop
`FUNCF`	`IFUNCF`	Fixed-arg Lua function
`FUNCV`	`IFUNCV`	Vararg Lua function

Table 4.8: Bytecode instructions to force interpretation

As mentioned, each time a trace aborts, the hotcode detected is penalised incrementing its *backoff* counter. The penalty mechanism uses a 64-entries table defined into the JIT state. Each row contains: (i) the starting bytecode PC of the hotpath; (ii) the penalty value (previously called *backoff* counter); (iii) the abort reason number (details in Tab. 4.6).

```
/* Round-robin penalty cache for bytecodes leading to aborted traces. */
typedef struct HotPenalty {
  MRef pc;    /* Starting bytecode PC. */
  uint16_t val;   /* Penalty value, i.e. hotcount start. */
  uint16_t reason;  /* Abort reason (really TraceErr). */
} HotPenalty;

#define PENALTY_SLOTS 64  /* Penalty cache slot. Must be a power of 2. */
#define PENALTY_MIN (36*2)  /* Minimum penalty value. */
#define PENALTY_MAX 60000 /* Maximum penalty value. */
#define PENALTY_RNDBITS 4 /* # of random bits to add to penalty value. */

/* JIT compiler state. */
typedef struct jit_State {
  ...
  HotPenalty penalty[PENALTY_SLOTS];  /* Penalty slots. */
  uint32_t penaltyslot; /* Round-robin index into penalty slots. */
  uint32_t prngstate; /* PRNG state. */
  ...
} jit_State;
```

Listing 4.7: `lj_traceerr.h`

`PENALTY_MIN` represents the minimum increment of the counter and `PENALTY_MAX` is the threshold for blacklisting. The variable `penaltyslot` is a round-robin index that points to the next available entry in the table. It is used when a new hotpath needs to be penalised. Tab. 4.9 shows a snapshot of a hot penalty table as an example.

Index	PC	val	reason
0	0x4157b014	55122	7
1	0x41635430	56340	7
2	0x41635070	1211	5
3	0x4157b03c	617	10
4	0	0	0
5	0	0	0
...	...	...	...
63	0	0	0

Table 4.9: Exemplifying snapshot of the Hot Penalty Table

The penalisation mechanism is the following. When a trace aborts, the function `trace_abort` is called. If the trace is a root trace, the PC of the starting

bytecode instruction is penalized (thus the hotcode is penalised). The function `penalty_pc` is responsible for the penalisation. If the penalty value exceeds the threshold (`PENALTY_MAX`), the trace is blacklisted. Thus, the bytecode is patched with the adequate `I...` operation and the hotcode previously detected can never become hot again. In LuaJIT, blacklisting is permanent. Once a bytecode is blacklisted, it can never be whitelisted.

A description of penalisation and blacklisting is shown in the flow chart at Fig. 4.9.

```
/* Penalize a bytecode instruction. */
static void penalty_pc(jit_State *J, GCproto *pt, BCIns *pc, TraceError e) {
  uint32_t i, val = PENALTY_MIN;
  for (i = 0; i < PENALTY_SLOTS; i++)
    if (mref(J->penalty[i].pc, const BCIns) == pc) {  /* Cache slot found? */
      /* First try to bump its hotcount several times. */
      val = ((uint32_t)J->penalty[i].val << 1) + LJ_PRNG_BITS(J, PENALTY_RNDBITS);
      if (val > PENALTY_MAX) {
        blacklist_pc(pt, pc);  /* Blacklist it, if that didn't help. */
        return;
      }
      goto setpenalty;
    }
  /* Assign a new penalty cache slot. */
  i = J->penaltyslot;
  J->penaltyslot = (J->penaltyslot + 1) & (PENALTY_SLOTS-1);
  setmref(J->penalty[i].pc, pc);
  setpenalty:
  J->penalty[i].val = (uint16_t)val;
  J->penalty[i].reason = e;
  hotcount_set(J2GG(J), pc+1, val);
}

/* Blacklist a bytecode instruction. */
static void blacklist_pc(GCproto *pt, BCIns *pc) {
  setbc_op(pc, (int)bc_op(*pc)+(int)BC_ILOOP-(int)BC_LOOP);
  pt->flags |= PROTO_ILOOP;
}
```

Listing 4.8: `lj_trace.c`

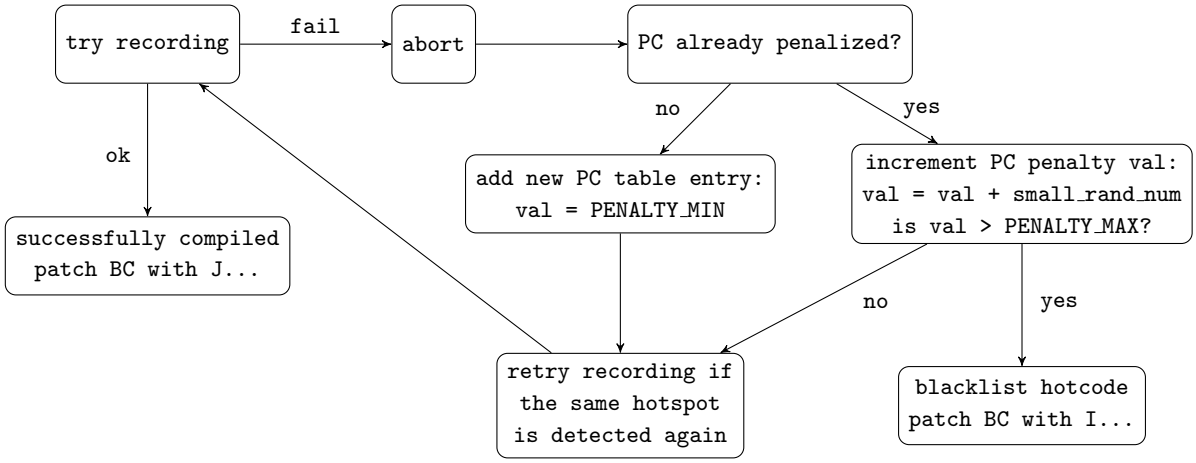

Figure 4.9: Penalization and blacklisting

Differently from the hotcount mechanism, collisions can never occur in the hot penalty table. The PC is used to check if a hotcode has already been penalized. Since the PC (that contains the memory address of the current bytecode instruction) is unique, collisions are impossible.

However, another small drawback can occur because of the round-robin index. When there are enough traces aborts after each other so that the hot penalty table is full (the 64 slots are already taken by ongoing penalised bytecode), a slot can be reused before its counter gets high enough to cause blacklisting. Thus, the next abort of the trace linked to the overwritten counter will be considered as its previous count was discarded. The trace compiler will link that hotpath to a different slot, as if it never aborted before.

The probability of ending up in such an inconvenient situation is relatively low. In fact, Mike Pall commented this issue on the LuaJIT mailing list saying: *"While theoretically possible, I've never seen this happen. In practice, something does get blacklisted eventually, which frees up a penalty tracking slot. There are only ever a handful of these slots in use. 64 slots is really generous"*[1].

Blacklisting cancer

Blacklisting is a very powerful tool that is used in the context of LuaJIT to save time avoiding to retry compilation of hotpaths that aborted repeatedly. However, a really unpleasant situation can arise as a consequence of the

[1]Conversation on the LuaJIT mailing list: `https://www.freelists.org/post/luajit/When-to-turn-off-JIT,4`

fact that in LuaJIT blacklisting is permanent. In other words, blacklisted fragments of code cannot be whitelisted ever again. On one hand, this is reasonable because there is no sense in recording a trace that contains a code fragment which failed to compile many times (hence it was blacklisted). On the other hand, it could be possible that the same code fragment can be successfully compiled in a different context of a new trace.

Moreover, LuaJIT blacklists hotpaths that hits an already blacklisted code fragment. To be more precise, if the interpreter hits an already blacklisted bytecode instruction (`I...`) while recording a new trace, trace creation is aborted and the new trace is also blacklisted. This could lead to a mechanism of *cascading blacklisting* from a code fragment to another. It is particularly dangerous when LuaJIT blacklists a key function of an application (i.e. a function called in many points of the source code) because the cascade of blacklisting spreads rapidly all over the code, hence the name *blacklisting cancer*.

To not fall is such an inconvenient situation we should avoid to use programming patterns which emphasise this problem. An exemplifying situation on this subject was discussed in the LuaJIT community[2]. This example is proposed again below to better illustrate the problem.

[2]Conversation on the LuaJIT mailing list: `https://www.freelists.org/post/luajit/ANN-dumpanalyze-tool-for-working-with-LuaJIT-dumps,12`

```lua
-- Blacklisting Cancer

-- Call a function with an argument.
local function apply (f, x)
  return f(x)
end

-- Return the value x wrapped inside a closure.
local function fwrap (x)
   return function () return x end -- Create a closure Not-Yet-Implemented (NYI)
end

-- Attempt to compile that naturally fails.
for i=1,1e5 do
  apply(fwrap, i) -- Abort message: NYI: create a closure
end

-- apply() function is now permanently blacklisted.
-- Every call to apply() in the future will have to be run by the interpreter.

local function nop (x)
end

-- Attempt to compile that should not fail.
for i=1,1e5 do
  apply(nop, i) -- Abort message: calling blacklisted function
end
```

The JIT permanently blacklists `apply()`. It will never allow that function to be called in a trace. In fact, the second loop could have been compiled successfully if it was run before the first loop.

It should be mention that other implementations of tracing JIT do not suffer from this problem. For instance, RaptorJIT [20] gives another chance to blacklisted code fragments of being compiled in a different context. In particular, when calling a blacklisted function from JIT code, it ignores the blacklisting and it just inline the content of the function. It tries to compile the function in the context of its caller. It can be worth to do it because sometimes code fails to compile in isolation, but a function can be compiled in the context of its caller. In fact, this implies overhead, since RaptorJIT puts more effort to generate machine code at the expense of making more compilation attempts.

4.6 Snapshots

Snapshot is a key feature used in LuaJIT to leave the VM in a consistent state when a trace exits. When the execution leaves a trace because of guard failures, the system switches to interpretation. The VM should be left in a consistent state for the interpret to continue. This means that all updates (stores) to the state (stack or objects) must track the original language semantics. In particular the values held in registers throughout the

trace must be written to their respective stack locations. Once the stack is in a suitable state, the interpreter can continue.

Within the possible techniques used in trace-based just-in-time compilation (see Sec. 2.3.6), LuaJIT solves the problem of keeping the VM in a consistent state by maintaining a mapping from stack location to SSA IR instructions. Such mappings are called *snapshots* in the LuaJIT codebase. Using a snapshot, LuaJIT can then reconstruct the operand stack as it would have been if the instructions in the trace were interpreted.

The Snapshot mechanism is implemented in the `lj_snap.[c,h]`, and the `SnapShot` data structure is defined in `lj_jit.h` (the key functions are `lj_snap_restore` and `snap_restoreval`).

Snapshots can be see at the IR level. From the definition in the official LuaJIT website [65]: a snapshot captures the IR references corresponding to modified slots and frames in the bytecode execution stack. Every snapshot saves a specific bytecode execution state, which can later be restored on trace exits. Snapshots are sparsely emitted and compressed. Snapshots provide the link between the IR and the bytecode domain (and transitively the source code domain, via the bytecode debug info).

Code sinking via snapshots allows sinking of arbitrary code without the overhead of the other approaches. A snapshot stores a consistent view of all updates to the state before an exit. If an exit is taken the on-trace machine state (registers and spill slots) and the snapshot can be used to restore the VM state.

The snapshot mechanism is slow if compared to other approaches such as *exit stubs* (see Sec. 2.3.6), but this should be seen as a problem in LuaJIT. By quoting Mike Pall in [66] *"State restoration using this data-driven approach is slow of course. But repeatedly taken side exits quickly trigger the generation of side traces. The snapshot is used to initialise the IR of the side trace with the necessary state using pseudo-loads. These can be optimised together with the remainder of the side trace. The pseudo-loads are unified with the machine state of the parent trace by the back-end to enable zero-cost linking to side traces"*.

4.6.1 Example

Here we propose a concrete example which describes snapshots.

```
1 -- Snapshot
2
3 local x = 0
4
5 -- A for loop implicitly define tacit variables:
```

```
6  -- index i (intern), limit N, step s, index i (extern)
7  for i=1,200 do
8    if i < 100 then x = x + 11
9    else x = x + 22 end
10 end
```

LuaJIT stores the local variables in stack slots according to the order in which they appear in the code. When a local variable is declared, a single slot is reserved to store it. On the other hand, a `for` loop reserves 4 slots that contain respectively: (i) the internal copy of the index, (ii) the limit of the loop, (iii) the step of each iteration and (iv) the external copy of the index. See Tab. 4.10 for details.

Stack slot number	Variable	Type	Structure
#0	-	-	closure/frame slot
#1	x	local var	declaration
#2	i	index (internal copy)	loop
#3	N	limit	
#4	s	step	
#5	i	index (external copy)	

Table 4.10: Slots variables allocation

When printing the IR through `jdump` you can visualise snapshots by selecting the flag 's'. A typical use of this option is shown below.

```
./luajit -jdump=is Ex.lua
```

Once you manage to print snapshots in the IR you should be aware that:

Each snaphot (SNAP) lists the modified stack slots and their values. The *i-th* value in the snapshot list represents the index of the IR that writes a value in slot number `#i`; "----" indicates that the slot is not written. Frames are separated by "|".

Going back to the example previously desbribe the corresponding IR is the following.

```
---- TRACE 1 start Ex.lua:7
---- TRACE 1 IR
....         SNAP   #0   [ ---- ]
0001    int SLOAD  #2    CI
0002 >  num SLOAD  #1    T
0003   + num ADD    0002  +11
0004   + int ADD    0001  +1
....         SNAP   #1   [ ---- 0003 ]
0005 >  int LE     0004  +100
....         SNAP   #2   [ ---- 0003 0004 ---- ---- 0004 ]
0006 ------ LOOP ------------
0007   + num ADD    0003  +11
```

```
0008  + int ADD     0004  +1
....        SNAP    #3    [ ---- 0007 ]
0009 >  int LE      0008  +100
0010    int PHI     0004  0008
0011    num PHI     0003  0007
---- TRACE 1 stop -> loop
```

In this example slot `#0` is not written since the frame slot doesn't change. In `SNAP #1` the value of x changes because we compute `x=x+11`, hence the IR at index `0003` writes a value in slot `#1`. `SNAP #2` stores the changes of x as before and it stores the value of `i` because it was incremented `i=i+1` and it is checked that `i<100`. In this case the IR at index `0004` writes a value in slot `#2` and `#5` (internal and external copy of the index). `SNAP #3` is the same of `SNAP #1` but it considers the IR at index `0007`.

4.6.2 Sparse snapshots

As previously mentioned when a guard fails the trace is exited and the VM must be in a consistent state. This is achieved through snapshots, but LuaJIT avoids generating a snapshot for each guard of a trace. Guards that have a low probability of being wrongly predicted do not cause snapshots. This is a feature of LuaJIT called *sparse snapshot* by Mike Pall. Here we propose his explanation about it presented in the LuaJIT 2.0 intellectual property disclosure and research opportunities [66].

"Taking a full snapshot of all state updates before every exit would need a considerable amount of storage. Since all scalar stores are sunk, it's feasible to reduce the snapshot density. The basic idea is that it doesn't matter which state is restored on a taken exit, as long as it's consistent.

This is a form of transactional state management. Every snapshot is a commit; a taken exit causes a rollback to the last commit. The on-trace state may advance beyond the last commit as long as this doesn't affect the possibility of a rollback. In practice this means that all on-trace updates to the state (non-scalar stores that are not sunk) need to force a new snapshot for the next exit.

Otherwise the trace recorder only generates a snapshot after control-flow constructs that are present in the source, too. Guards that have a low probability of being wrongly predicted do not cause snapshots (e.g. function dispatch). This further reduces the snapshot density. Sparse snapshots also improve on-trace code quality, because they reduce the live range of the results of intermediate computations. Scheduling decisions can be made over a longer stream of instructions, too".

4.7 Variables allocation

The aim of this section is to describe how variables are allocated in the bytecode and in the SSA IR. In particular, (i) the first paragraph investigates the allocation of local variables, (ii) the second paragraph considers the case of global variables and (iii) the last paragraphs examines upvalues. In each of them it will be shown an example that clarifies what explained.

4.7.1 Local variables

Local variables are stored in stack slots according to the order they appear in the code. When a local variable is declared, a single slot is reserved to store it. On the other hand, a `for` loop reserves 4 slots that contain respectively: (i) the internal copy of the index, (ii) the limit of the loop, (iii) the step of each iteration and (iv) the external copy of the index.

```
1  -- Nested loop with local variables
2
3  local iN, iS = 1e3, 2   -- i limit, step
4  local jN, jS = 1e4, 3   -- j limit, step
5
6  for i=1,iN,iS do        -- outer loop
7    local x = 0
8    for j=1,jN,jS do      -- inner loop
9      i = i + 1
10     x = x + i
11   end
12 end
```

The example above contains in order: variables declaration (line `4`), for loop (line `6`), variable declaration (line `7`), for loop (line `8`). Thus, variables will be allocated in order in the stack slots as show in table 4.11.

Structure	Type	Variable	Stack slot number
declaration	local var	iN	#1
	local var	iS	#2
	local var	jN	#3
	local var	jS	#4
outer loop	index (internal copy)	i	#5
	limit	iN	#6
	step	iS	#7
	index (external copy)	i	#8
declaration	local var	x	#9
inner loop	index (internal copy)	j	#10
	limit	jN	#11
	step	jS	#12
	index (external copy)	j	#13

Table 4.11: Local variables allocation

Below it is shown the IR dump. Variables are loaded with the `SLOAD` (stack slot load) operation where the left operand `#`n refers to the variable slot number and the right operand contains flags (see **??** for details about flags).

```
---- TRACE 1 start Ex.lua:8
---- TRACE 1 IR
0001 rax    >  int SLOAD  #11   CRI
0002 rcx    >  int SLOAD  #12   CRI
0003        >  int GE     0002  +0
0004 rbx    >  int ADDOV  0002  0001
0005           int USE    0004
0006 rbp       int SLOAD  #10   CI
0007 xmm6   >  num SLOAD  #8    T
0008 xmm6    + num ADD    0007  +1
0009 xmm7   >  num SLOAD  #9    T
0010 xmm7    + num ADD    0009  0008
0011 rbp     + int ADD    0006  0002
0012        >  int LE     0011  0001
0013 ------------ LOOP ------------
0014 xmm6    + num ADD    0008  +1
0015 xmm7    + num ADD    0014  0010
0016 rbp     + int ADD    0011  0002
0017        >  int LE     0016  0001
0018 rbp       int PHI    0011  0016
0019 xmm6      num PHI    0008  0014
0020 xmm7      num PHI    0010  0015
```

```
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/3 Ex.lua:6
---- TRACE 2 IR
0001 xmm6      num SLOAD  #8    PI
0002 xmm7      num SLOAD  #9    PI
0003 xmm2      num SLOAD  #6    RI
0004 xmm1      num SLOAD  #7    RI
0005        >  num GE     0004  +0
0006 xmm7      num SLOAD  #5    I
0007 xmm7      num ADD    0006  0004
0008        >  num LE     0007  0003
0009 xmm4   >  num SLOAD  #3    T
0010 xmm3   >  num SLOAD  #4    T
0011 rbp    >  int CONV   0009  int.num
0012 rbx    >  int CONV   0010  int.num
0013        >  int GE     0012  +0
0014 rbx    >  int ADDOV  0012  0011
0015           int USE    0014
0016        >  int GE     0011  +1
---- TRACE 2 stop -> 1
```

Note that `#0` indicates the closure/frame slot and `#1` the first variable slot (corresponding to slot `0` of the bytecode). Moreover, there are no store operations for stack slots. When the execution flow exits from a trace, values on stack slots are restored. All stores to stack slots are effectively sunk into exits or side traces [65].

4.7.2 Global variables

Dealing with global variables is more subtle because their value can be modified from different part of the code. In fact, the concept of global variables does not exist explicitly in Lua. To preserve the illusion of global variables, Lua keeps them internally in a regular table that is used as *global environment* (this is a simplification of what really occurs, see [67] for more details).

```
1 -- Loop with global variables
2
3 x, y = 0, 0
4
5 for i=1,1e3 do
6   x = x + 11
7   y = 22
8 end
```

The example contains two global variables `x` and `y`. The value of `x` is updated at each iteration of the loop, while `y` can be moved outside the body of the loop by LICM without affecting the semantic of the program. As expected this optimisation is performed only at IR level, but not in the bytecode.

```
---- TRACE 1 start Ex.lua:5
0009  GGET     4   1      ; "x"
0010  ADDVN    4   4   0  ; 11
0011  GSET     4   1      ; "x"
0012  KSHORT   4   22
0013  GSET     4   0      ; "y"
0014  FORL     0 => 0009
---- TRACE 1 stop -> loop
```

As shown in the bytecode above, `x` and `y` are retrieved and modified with the instructions `GGET` (global get) and `GSET` (global set) because they are global variables.

In the IR it is possible to see more details of what really occurs.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001 rbp       int SLOAD  #1    CI
0002 rbx       fun SLOAD  #0    R
0003 rcx       tab FLOAD  0002  func.env
0004           int FLOAD  0003  tab.hmask
0005       >   int EQ     0004  +63
0006 rdx       p32 FLOAD  0003  tab.node
0007 rax   >   p32 HREFK  0006  "x"  @33
0008 xmm7  >   num HLOAD  0007
0009 xmm7   +  num ADD    0008  +11
0010           num HSTORE 0007  0009
0011       >   p32 HREFK  0006  "y"  @34
0012           tab FLOAD  0003  tab.meta
0013       >   tab EQ     0012  NULL
0014           num HSTORE 0011  +22
0015           nil TBAR   0003
0016 rbp    +  int ADD    0001  +1
0017       >   int LE     0016  +1000
0018 ------------ LOOP ------------
0019 xmm7   +  num ADD    0009  +11
0020           num HSTORE 0007  0019
0021 rbp    +  int ADD    0016  +1
0022       >   int LE     0021  +1000
0023 rbp       int PHI    0016  0021
0024 xmm7      num PHI    0009  0019
---- TRACE 1 stop -> loop
```

`SLOAD` at line `0001` refers to the variable `i` and `SLOAD` at line `0002` is relative to the starting frame of the trace, where `#0` indicates the frame slot.

Then, FLOAD (object field load) at line 0003 accesses the global environment (func.env). At line 0004 the size of the hash part (tab.hmask) is loaded and at line 0005 it is checked that its value is not changed (hmask=+63). Finally, the table hash part (tab.node) is loaded at line 0006 and with HREFK (constant hash reference) "x" is retrieved from the global environment table (line 0007). Once the reference has been retrieved the value of x is loaded with HLOAD (line 0008) and stored with HSTORE (line 0010 and 0020).

Since the global environment has already been loaded, there is no need to do it again for y. Thus, "y" can be retrieved with HREFK (line 0011). Then, the global environment metatable is loaded (line 0012) and it is checked that it is equal to NULL (line 0013). After storing the value of y with HSTORE (line 0014) the compiler creates a table write barrier (line 0015). TBAR (table write barrier) is a write barrier needed for the incremental garbage collector.

4.7.3 Upvalues

Closure and upvalues are a key concept in Lua. Thus, LuaJIT has specific instructions to handle them both at bytecode and IR level.

The example below shows a loop with a closure that contains an upvalue: the closure is inlined in the trace and the upvalue is modified inside it.

```
1  -- Recursive function with upvalue
2
3  local function f(x)
4    return function(y)    -- return a closure
5      x = x + 1
6      return x + y
7    end
8  end
9
10 local a = 0
11 local b = f(0)
12
13 for i=1,1e3 do
14   a = b(i) + 2
15 end
```

In the bytecode below it is possible to see the fact that the closure is inlined (lines 0000-00006). The dot in the second column of the bytecode represents the depth in the call. What is interesting is the fact that the upvalue x is retrieved and modified with the instructions UGET (get upvalue) and USETV (set upvalue to variable).

```
---- TRACE 1 start Ex.lua:13
0010  MOV       7   2
0011  MOV       8   6
0012  CALL      7   2   2
0000  . FUNCF     2           ; Ex.lua:4
0001  . UGET      1   0       ; x
0002  . ADDVN     1   1   0   ; 1
0003  . USETV     0   1       ; x
0004  . UGET      1   0       ; x
0005  . ADDVV     1   1   0
0006  . RET1      1   2
0013  ADDVN     1   7   0   ; 2
0014  FORL      3 => 0010
---- TRACE 1 stop -> loop
```

The same mechanism is shown in the IR.

```
---- TRACE 1 start Ex.lua:13
---- TRACE 1 IR
0001 rbp       int SLOAD  #4     CI
0002        >  fun SLOAD  #3     T
0003        >  fun EQ     0002   Ex.lua:4
0004        >  p32 UREFC  Ex.lua:4  #0
0005 xmm6   >  num ULOAD  0004
0006 xmm6    + num ADD    0005   +1
0007           num USTORE 0004   0006
0008 xmm7      num CONV   0001   num.int
0009 xmm7      num ADD    0008   0006
0010 xmm7    + num ADD    0009   +2
0011 rbp     + int ADD    0001   +1
0012        >  int LE     0011   +1000
0013 ------------ LOOP ------------
0014 xmm6    + num ADD    0006   +1
0015           num USTORE 0004   0014
0016 xmm7      num CONV   0011   num.int
0017 xmm7      num ADD    0016   0014
0018 xmm7    + num ADD    0017   +2
0019 rbp     + int ADD    0011   +1
0020        >  int LE     0019   +1000
0021 rbp       int PHI    0011   0019
0022 xmm6      num PHI    0006   0014
0023 xmm7      num PHI    0010   0018
---- TRACE 1 stop -> loop
```

`SLOAD` at line `0001` refers to the variable `i` and `SLOAD` at line `0002` loads the closure assigned to the variable `b`. Then it is checked that the closure loaded is equal to what is returned at `Ex.lua:4` (line `0003`). `UREFC` (closed upvalue reference) gets the upvalue reference from the closure (line `0004`). Finally, it is possible to load and store the upvalue with `ULOAD` (line `0005`) and `USTORE` (line `0007` and `0015`).

Chapter 5

Side traces and Stitch traces

The aim of this chapter is to show some concrete experimental cases in order to understand how multiple traces are generated and organised by LuaJIT. Multiple traces appear when there is a branch in the control flow and more than one path is frequently executed. In this case, LuaJIT will create a root trace with *side traces* attached to it.

Another important aspect that will be explored is *stitch traces*. It is a mechanism that avoids trace aborts due to Not-Yet-Implemented (NYI) functions when encountering a C function or not-compiled built-in.

Before to investigate these aspects, we will present canonical transformations of logical expressions, loops equivalence and asserts. These structures create branches in the code that can be represented by simple if-statements and for loops. This is the reason why in the second part of this chapter we only will consider if-statements and for loops to investigate how LuaJIT creates *side traces* and *stitch trace*.

Concerning *side traces*, we will first clarifies how the just-in-time compiler (JIT) generate traces from simple loops. Then we will analyse how traces are connected with each other in more complex structures. Finally, we will investigates recursive functions. For each case it is described how the compiler behaves and it is shown: the LUA code of the example; the corresponding bytecode and intermediate representation (IR) generated; a flow diagram that refers to the IR.

5.1 Canonical transformations

This section aims to analyse the traces generated by LuaJIT for logical expressions and their corresponding if-statements transformation. Then, it will present the equivalence between possible loop structures in Lua and how they

are mapped to traces by the JIT compiler. Finally, it will explore the JIT behaviour when dealing with asserts.

5.1.1 Logical transformations

Lua logical operators

Before to introduce how LuaJIT generate traces when dealing with logical expression, we must clarify the rationale behind logical operators in Lua. Here we propose again the explanation taken from the reference manual [68].

The logical operators in Lua are `and`, `or`, and `not`. Like the control structures, all logical operators consider both `false` and `nil` as `false` and anything else as `true`.

The negation operator `not` always returns `false` or `true`. The conjunction operator `and` returns its first argument if this value is `false` or `nil`; otherwise, and returns its second argument. The disjunction operator `or` returns its first argument if this value is different from `nil` and `false`; otherwise, or returns its second argument. Both `and` and `or` use short-cut evaluation; that is, the second operand is evaluated only if necessary.

Trace creation with logical operators

As explained in the previous chapters LuaJIT generates traces from recording the executed bytecode instructions of hotpaths (either hotloop or hotfunctions). The high-level structure of the Lua code plays a key role to determine which bytecode instructions are executed because it determines which branch of the control flow is taken.

The rationale applied by LuaJIT in generating traces from standards control structures (e.g. if-statements) is exactly the same when dealing with logical operators.

Once understood the JIT behaviour for hotpaths which contain control structures (e.g. if-statements) you can apply the same rationale to logical operators because every expressions with logical operators can be transformed to the corresponding if-statement expressions.

In this section we present the canonical transformation of logical operators in the corresponding if-statements expressions. We illustrate concrete examples of the logical operators `and` and `or` considering all the possible combinations of the operands values (`true`, `false`, `nil`).

And

```
-- And

local a, b, c = v1, v2, nil -- v1, v2 in (true, false, nil)

for i=1,100 do
  c = a and b
end
```

```
-- And with if-statement

local a, b, c = v1, v2, nil -- v1, v2 in (true, false, nil)

for i=1,100 do
  if a then c = b else c = a end
end
```

c	a	b
2	1	2
false	1	false
nil	1	nil
false	false	2
false	false	false
false	false	nil
nil	nil	2
nil	nil	false
nil	nil	nil

Table 5.1: Operands values of `c = a and b`

Or

```
-- Or

local a, b, c = v1, v2, nil -- v1, v2 in (true, false, nil)

for i=1,100 do
  c = a or b
end
```

```
-- Or with if-statement

local a, b, c = v1, v2, nil -- v1, v2 in (true, false, nil)

for i=1,100 do
```

```
  if a then c = a else c = b end
end
```

c	a	b
1	1	2
1	1	false
1	1	nil
2	false	2
false	false	false
nil	false	nil
2	nil	2
false	nil	false
nil	nil	nil

Table 5.2: Operands values of `c = a or b`

And-Or

```
-- And-Or

local a, b, c, d = v1, v2, v3, nil -- v1, v2, v3 in (true, false, nil)

for i=1,100 do
  d = a and b or c
end
```

```
-- And-Or with if statements

local a, b, c, d = v1, v2, v3, nil -- v1, v2, v3 in (true, false, nil)

for i=1,100 do
  if a then
    if b then d = b else d = c end
  else
    d = c
  end
end
```

d	a	b	c
2	1	2	3
2	1	2	false
2	1	2	nil
3	1	false	3
false	1	false	false
nil	1	false	nil
3	1	nil	3
false	1	nil	false
nil	1	nil	nil
3	false	2	3
false	false	2	false
nil	false	2	nil
3	false	false	3
false	false	false	false
nil	false	false	nil
3	false	nil	3
false	false	nil	false
nil	false	nil	nil
3	nil	2	3
false	nil	2	false
nil	nil	2	nil
3	nil	false	3
false	nil	false	false
nil	nil	false	nil
3	nil	nil	3
false	nil	nil	false
nil	nil	nil	nil

Table 5.3: Operands values of `d = a and b or c`

Or-And

```
-- Or-And

local a, b, c, d = v1, v2, v3, nil -- v1, v2, v3 in (true, false, nil)

for i=1,100 do
  d = a or b and c
end
```

```lua
-- Or-And with if statements

local a, b, c, d = v1, v2, v3, nil -- v1, v2, v3 in (true, false, nil)

for i=1,100 do
    if a then d = a
    elseif b then d = c
    else d = b end
end
```

d	a	b	c
1	1	2	3
1	1	2	false
1	1	2	nil
1	1	false	3
1	1	false	false
1	1	false	nil
1	1	nil	3
1	1	nil	false
1	1	nil	nil
3	false	2	3
false	false	2	false
nil	false	2	nil
false	false	false	3
false	false	false	false
false	false	false	nil
nil	false	nil	3
nil	false	nil	false
nil	false	nil	nil
3	nil	2	3
false	nil	2	false
nil	nil	2	nil
false	nil	false	3
false	nil	false	false
false	nil	false	nil
nil	nil	nil	3
nil	nil	nil	false
nil	nil	nil	nil

Table 5.4: Operands values of `d = a or b and c`

And with function

```lua
-- And with function

local a,b = v1, nil -- v1 in (1, 2, 3, false, nil)

local function f(a)
  if a == 1 then return true
  elseif a == 2 then return false
  else return nil end
end

for i=1,100 do
  b = a and f(a) or nil
end
```

```lua
-- And with function if-statements

local a,b = v1, nil -- v1 in (1, 2, 3, false, nil)

local function f(a)
  if a == 1 then return true
  elseif a == 2 then return false
  else return nil end
end

for i=1,100 do
  if a then
    local tmp = f(a)
    if tmp then b = tmp else b = nil end
  else
    b = nil
  end
end
```

b	a	f(a)	nil
true	1	true	nil
nil	2	false	nil
nil	3	nil	nil
nil	false	-	nil
nil	nil	-	nil

Table 5.5: Operands values of `b = a and f(a) or nil`

5.1.2 Loops equivalence

In this section we present the equivalence between different loop structures in Lua and how they are handled by LuaJIT when compiling a trace. For each example we show the recorded bytecode instructions and the IR (the details of the syntax will be explained in the next section). When investigating the equivalence between code structures we must especially focus on the IR because it will be directly mapped to the compiled machine code.

Numeric for loop

The numeric `for` loop is the most simple and intuitive way to define a loop in Lua.

```
1 -- Numeric for loop
2
3 local x = 0
4
5 for i=1,100 do
6   x = x + 11
7 end
```

The recorded bytecode instructions are the following.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   0  ; 11
0007  FORL     1 => 0006
---- TRACE 1 stop -> loop
```

Looking at the IR you can notice that the index variable `i` is considered by LuaJIT as an integer value (line `0001`).

```
---- TRACE 1 start Ex.lua:5
0001    int SLOAD  #2    CI
0002 >  num SLOAD  #1    T
0003  + num ADD    0002  +11
0004  + int ADD    0001  +1
0005 >  int LE     0004  +100
0006 ------ LOOP ------------
0007  + num ADD    0003  +11
0008  + int ADD    0004  +1
0009 >  int LE     0008  +100
0010    int PHI    0004  0008
0011    num PHI    0003  0007
---- TRACE 1 stop -> loop
```

While loop

The same loop previously presented can be realised by a `while` structure.

```
1 -- While loop
2
3 local x, i = 0, 1
4
5 while i <= 100 do
6   x = x + 11
7   i = i + 1
```

```
8 end
```

The recorded bytecode instructions are the following.

```
---- TRACE 1 start Ex.lua:5
0007  ADDVN    0   0   0  ; 11
0008  ADDVN    1   1   1  ; 1
0009  JMP      2 => 0003
0003  KSHORT   2 100
0004  ISGT     1   2
0005  JMP      2 => 0010
0006  LOOP     2 => 0010
---- TRACE 1 stop -> loop
```

In this case the index variable `i` is considered as a numeric value in the IR (line `0001`) because it can be changed by the user at any time in the loop body.

```
---- TRACE 1 start Ex.lua:5
0001 >  num SLOAD  #1    T
0002  + num ADD    0001  +11
0003 >  num SLOAD  #2    T
0004  + num ADD    0003  +1
0005 >  num LE     0004  +100
0006 ------ LOOP ------------
0007  + num ADD    0002  +11
0008  + num ADD    0004  +1
0009 >  num LE     0008  +100
0010    num PHI    0002  0007
0011    num PHI    0004  0008
---- TRACE 1 stop -> loop
```

Repeat until loop

The `repeat until` structure is equivalent to the onces previously presented, but the first iteration of the loop is always executed. From the point of view of the JIT this does not change the result (in terms of generated traces) because a loop is recorded only if its iteration frequency exceeds a certain threshold.

```
1 -- Repeat until loop
2
3 local x, i = 0, 1
4
5 repeat
6   x = x + 11
7   i = i + 1
8 until i > 100
```

The recorded bytecode instructions are the following.

```
---- TRACE 1 start Ex.lua:5
0004  ADDVN    0   0   0  ; 11
0005  ADDVN    1   1   1  ; 1
0006  KSHORT   2 100
0007  ISGE     2   1
0008  JMP      2 => 0003
0003  LOOP     2 => 0009
---- TRACE 1 stop -> loop
```

The IR is identical to the `while` structure with the only difference that the end-condition of the loop is `i>100` (either than `i≤100`). The IR instruction is `ISGE` "is grater than" at line `0007`. This is because the loop repeats until the given condition becomes true.

```
---- TRACE 1 start Ex.lua:5
0001 >  num SLOAD  #1    T
0002  + num ADD    0001  +11
0003 >  num SLOAD  #2    T
0004  + num ADD    0003  +1
0005 >  num ULE    0004  +100
0006 ------ LOOP ------------
0007  + num ADD    0002  +11
0008  + num ADD    0004  +1
0009 >  num ULE    0008  +100
0010    num PHI    0002  0007
0011    num PHI    0004  0008
---- TRACE 1 stop -> loop
```

Generic for loop

Generic `for` is a Lua structure that allows you to traverse all values returned by an iterator function. Its syntax is the following.

```
1 for vars... in iter,state,ctl do body end
```

A generic loop cannot be compiled if the iterator uses the built-in function `next()` (e.g. `pairs()`) because LuaJIT does not support the compilation of `next()`. In fact, recording aborts if `next()` is executed because it belongs to the Not-Yet-implemented features of the JIT compiler.

On the other hand, a generic loop not based on `next()` (e.g. `ipairs()`) can be compiled. Here we propose an example.

```
1 -- Ipairs loop
2
3 local x, array = 0, {}
4 for i=1,100 do array[i] = i end
5
6 for i,v in ipairs(array) do
7   x = x + v + 11
8 end
```

The recorded bytecode instructions are the following. Note that `TRACE 1` is not shown, but it simply refers to the numeric loop at line 4.

```
---- TRACE 2 start Ex.lua:6
0013  ADDVV    7   0   6
0014  ADDVN    0   7   0   ; 11
0015  ITERC    5   3   3
0000  . FUNCC           ; ipairs_aux
0016  ITERL    5 => 0013
---- TRACE 2 stop -> loop
```

The overall IR is similar to the other loop structures previously presented. The main differences consists of handling the table. The end-condition of the loop is represented by the `ABC` instruction (Array Bounds Check: bound $>$ index) at lines `0012`, `0021`.

```
---- TRACE 2 start Ex.lua:6
---- TRACE 2 IR
0001 >  num SLOAD  #1    T
0002 >  num SLOAD  #7    T
0003    num ADD    0002  0001
0004  + num ADD    0003  +11
0005 >  fun SLOAD  #3    T
0006 >  tab SLOAD  #4    T
```

```
0007 >  num SLOAD  #5    T
0008 >  fun EQ     0005  ipairs_aux
0009    int CONV   0007  int.num
0010  + int ADD    0009  +1
0011    int FLOAD  0006  tab.asize
0012 >  int ABC    0011  0010
0013    p32 FLOAD  0006  tab.array
0014    p32 AREF   0013  0010
0015 >+ num ALOAD  0014
0016 ------ LOOP ------------
0017    num ADD    0015  0004
0018  + num ADD    0017  +11
0019    num CONV   0010  num.int
0020  + int ADD    0010  +1
0021 >  int ABC    0011  0020
0022    p32 AREF   0013  0020
0023 >+ num ALOAD  0022
0024    num PHI    0004  0018
0025    num PHI    0015  0023
0026    int PHI    0010  0020
---- TRACE 2 stop -> loop
```

5.1.3 Assert

An assert in Lua is treated by LuaJIT as a "standard" if-statement condition. The example below shows a case were the assert becomes false after 100 iteration of the loop.

```lua
-- Assert

local assert = assert

for i=1,200 do
  assert(i<100, "Case failed: expected i<100 but got instead.")
end
```

In this case the JIT will create a trace with a guard condition on '`i<100`' (line `0003`, `0008` in the IR), which verifies that its value is less than 100. The details of the syntax of the IR are explained from the next paragraph.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001    int SLOAD  #3    CI
0002 >  fun SLOAD  #2    T
0003 >  int LT     0001  +100
0004 >  fun EQ     0002  assert
0005  + int ADD    0001  +1
0006 >  int LE     0005  +200
0007 ------ LOOP ------------
0008 >  int LT     0005  +100
0009  + int ADD    0005  +1
0010 >  int LE     0009  +200
0011    int PHI    0005  0009
---- TRACE 1 stop -> loop
```

5.2 Essential cases

This section illustrates how the compiler creates traces in simple but significant cases: (i) empty loop, (ii) loop with assignment, (iii) loop with if-statement and (iv) nested loop.

5.2.1 Empty loop

Even an empty loop can generate a trace. When the loop becomes hot, the virtual machine (VM) starts to record the instructions and the types of their

operands during execution. Thus, it generates the equivalent IR.

```
1 -- Empty loop
2 for i=1,100 do
3
4 end
```

In this case the bytecode produced contains just the `FORL` loop instruction.

```
---- TRACE 1 start Ex.lua:3
0005  FORL     0 => 0005
---- TRACE 1 stop -> loop
```

On the other hand, the IR holds more interesting information because it shows the fact that the first iteration of the loop is unrolled.

```
---- TRACE 1 start Ex.lua:3
---- TRACE 1 IR
0001    int SLOAD  #1    CI
0002  + int ADD    0001  +1
0003 >  int LE     0002  +100
0004 ------ LOOP ------------
0005  + int ADD    0002  +1
0006 >  int LE     0005  +100
0007    int PHI    0002  0005
---- TRACE 1 stop -> loop
```

The instruction at the first line `SLOAD` (stack slot load) is used to init the variable i used by the loop where its left operand #1 refers to the first variable slot and the right operand contains two flags: coalesce (C) and inherited (I).

The next lines are supposed to contain the loop, but the same instructions are repeated twice. This is due to the fact that the first iteration of the loop is unrolled (lines `0002-0003`), then the actual loop (lines `0005-0006`) is shown after the `-- LOOP --` label (line `0004`). The first iteration ensures that preconditions for all subsequent instructions are met. `ADD` increments the loop counter i and `LE` (left operand $\leq$ right operand) checks that its value is lower than 100. If this condition is not satisfied ($i > 100$), the execution takes the trace exit at line `0003` or at line `0006`. Possible exits from the trace are indicated by the symbol $>$ in the second column of the instruction. If the condition is true ($i \leq 100$) the execution flow makes a backward jump to the `-- LOOP --` label (line `0004`) and continues with the next iteration of the loop. It is important to highlight the fact that only the instructions at lines `0005-0006` are executed repeatedly.

Eventually, the `PHI` instruction positioned at the end of the looping trace (line `0007`) allows to select values from different incoming path at control flow merge points [69]. The left operand `0002` holds a reference to the initial value of i, the right operand `0005` holds a reference to the value after each loop iteration. Operands of the PHI function are indicated by the symbol $+$ in the second column of the IR (in this case lines `0002` and `0005`).

The diagram below explains the execution flows of the IR in a cleaner way. Specially for the next complex examples it will be easier to look at the diagram to understand the IR.

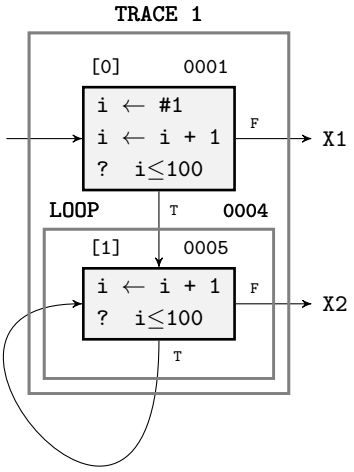

Figure 5.1: Trace flow diagram empty loop

In the diagrams each trace is divided into blocks containing instructions with a unique identifier enclosed in squared brackets (e.g. `[0]`). On the top right of each block it is indicated the line in the IR of the first instruction in the block (e.g. `0001`). At the end of each block there could be a conditional expression that represents a guard. In the case that the guard is violated (the condition is false) the trace is exited, otherwise the execution continues to the next block of the trace. Possible exits are represented by the letter 'X' followed by their number (e.g. X1). By default an exit leads to the virtual machine. Note that when the execution flow exits from a trace, values on stack slots are restored.

5.2.2 Loop with assignment

This example has been designed to analyse what happens if the loop contains variable assignments. It will be shown that the compiler is able to move invariant instructions out of loops[1] with the loop-invariant code motion (LICM) optimisation. Another small difference from the previous example is that the maximum loop counter is not a literal but it is a variable ($N = 100$).

[1]Conversation on the LuaJIT mailing list: `https://www.freelists.org/post/luajit/how-to-understand-the-structure-of-irmcode-dump,1`

```
1  -- Loop with assignment
2
3  local x = 0
4  local y = 0
5  local N = 100
6
7  for i=1,N do
8     y = 11
9     x = x + 22
10 end
```

As shown in the bytecode below, the instruction `KSHORT` (line `0008`) sets y to 11 and `ADDVN` (line `0009`) computes the operation $x = x + 22$. Here it is clear that at bytecode level the LICM optimisation is not applied because the execution flow makes a backward jump to line `0008` and $y = 11$ is repeated at each iteration of the loop. In fact, no optimisation is performed by LuaJIT on bytecode.

```
---- TRACE 1 start Ex.lua:7
0008  KSHORT   1  11
0009  ADDVN    0   0   0  ; 22
0010  FORL     3 => 0008
---- TRACE 1 stop -> loop
```

It should be noted that the initialisation of variable values are performed outside traces because these instruction are not executed repeatedly, but they are executed just once.

In the IR below it is possible to see more details of what really occurs.

```
---- TRACE 1 start Ex.lua:7
---- TRACE 1 IR
0001 >  int SLOAD  #5    CRI
0002 >  int LE     0001  +2147483646
0003    int SLOAD  #4    CI
0004 >  num SLOAD  #1    T
0005  + num ADD    0004  +22
0006  + int ADD    0003  +1
0007 >  int LE     0006  0001
0008 ------ LOOP ------------
0009  + num ADD    0005  +22
0010  + int ADD    0006  +1
0011 >  int LE     0010  0001
0012    int PHI    0006  0010
0013    num PHI    0005  0009
---- TRACE 1 stop -> loop
```

The first two lines refer to the maximum loop counter N: `SLOAD` (line `0001`) with flag read-only (R) is used to init N and in line `0002` it is checked if its value falls into the signed 32-bit integer range ($N \leq +2147483646$). In this way, the compiler can discriminate if the loop will be done over integer or floating point values. The `SLOAD`s at lines `0003`-`0004` are used to init the variables i and x respectively. In particular x has a flag of type check (T).

At IR level it is possible to see compiler optimisations. The value of x changes at each iteration of the loop. Thus, the `ADD` instruction $x = x + 22$ is contained both in the pre-loop (line `0005`) and in the actual loop (line `0009`). On the other hand, the expression $y = 11$ can be moved outside the body of the loop by LICM without affecting the semantics of the program. This

instruction will be executed only once outside the trace (in fact there is no line in the IR referring to it).

At the end of the dump there are two `PHI` functions. The first (line `0012`) refers to the variable i as explained in the previous example. The second (line `0013`) refers to the variable x and it is necessary for the same reason.

The graph below shows what was just explained.

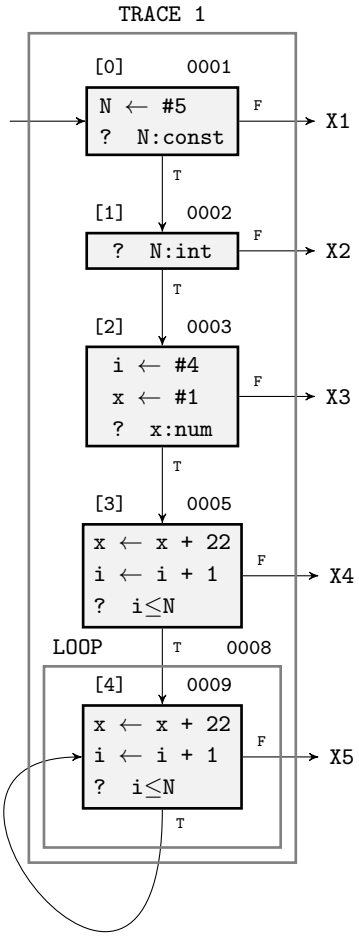

Figure 5.2: Trace flow diagram loop with assignment

5.2.3 Loop with if-statements

In this example the goal is to investigate how the compiler creates traces when an if-statement is contained inside a loop.

```
1 -- Loop with if-statement
2
```

```
3  local x = 0
4
5  for i=1,1e4 do
6    x = x + 11
7    if i%10 == 0 then    -- if-statement
8      x = x + 22
9    end
10   x = x + 33
11 end
```

In the loop shown above, the execution flow skips most of the times the instruction contained in the if-statement because the expression $i\%10 == 0$ is true only every 10 iterations of the loop. From $i = 0$ forward, the instructions that are executed repeatedly the most are $x = x + 11$ and $x = x + 33$, thus the compiler creates a trace containing these instructions (`TRACE 1`). By increasing i, the condition of the if-statement becomes true more and more often, thus the compiler will generate a side trace (`TRACE 2`) that contains the instruction within the if-statement and what follows down to the loop "end".

The bytecode below shows more in details this method (these are the bytecode instructions recorded by the JIT).

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   0  ; 11
0007  MODVN    5   4   1  ; 10
0008  ISNEN    5   2      ; 0
0009  JMP      5 => 0011
0011  ADDVN    0   0   4  ; 33
0012  FORL     1 => 0006

---- TRACE 1 stop -> loop

---- TRACE 2 start 1/4 Ex.lua:8
0010  ADDVN    0   0   3  ; 22
0011  ADDVN    0   0   4  ; 33
0012  JFORL    1   1
---- TRACE 2 stop -> 1
```

The expression $x = x + 11$ is computed at line `0006`. Then in line `0007` it is calculated $i\%10$ and in line `0008` it is checked that the result is not equal to zero. If the condition is true the execution flow jumps to the instruction at line `0011` where $x = x + 33$ is computed and then line `0012` contains the loop backward branch. If the condition is false ($i\%10 = 0$) the `JMP` to 0011 is not taken. The execution flow goes from the instruction at line `0009` to the very next instruction at line `0010`. In fact, this is the link between the root trace (`TRACE 1`) and the side trace (`TRACE 2`). Finally, in the side trace both the `ADD` operations $x = x + 22$ and $x = x + 33$ are executed. To conclude the execution flow goes back to the parent trace `-- TRACE 2 stop -> 1`.

The IR follows the same logic. What changes is the fact that the first iteration of the loop is unrolled. Moreover, in the IR it is more clear how the two traces are connected to each other. At the very first line of the side trace `-- TRACE 2 start 1/4` the number 1 refers to the parent trace and 4 to the exit number that corresponds to line `0012` in `TRACE 1`. Line `0012` is

the exit number 4, which is the 4th line having as second column the symbol > in `TRACE 1`.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001     int SLOAD  #2    CI
0002 >   num SLOAD  #1    T
0003     num ADD    0002  +11
0004     int MOD    0001  +10
0005 >   int NE     0004  +0
0006   + num ADD    0003  +33
0007   + int ADD    0001  +1
0008 >   int LE     0007  +10000
0009 ------ LOOP ------------
0010     num ADD    0006  +11
0011     int MOD    0007  +10
0012 >   int NE     0011  +0
0013   + num ADD    0010  +33
0014   + int ADD    0007  +1
0015 >   int LE     0014  +10000
0016     int PHI    0007  0014
0017     num PHI    0006  0013
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/4 Ex.lua:8
---- TRACE 2 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +22
0004     num ADD    0003  +33
0005     int ADD    0002  +1
0006 >   int LE     0005  +10000
0007     num CONV   0005  num.int
---- TRACE 2 stop -> 1
```

The diagram in Fig. 5.3 shows the IR flow diagram for this example. `TRACE 1` is organised as follow: blocks `[0]`,`[1]`,`[2]` contain the first pass of the unrolled loop; blocks `[3]`,`[4]` contain the *n-1* iterations of the actual loop; in block `[3]` there is a possible exit that leads to the side trace. At the end, when `TRACE 2` is finished, the execution flow joins `TRACE 1` in block `[0]`, while other exits join the VM.

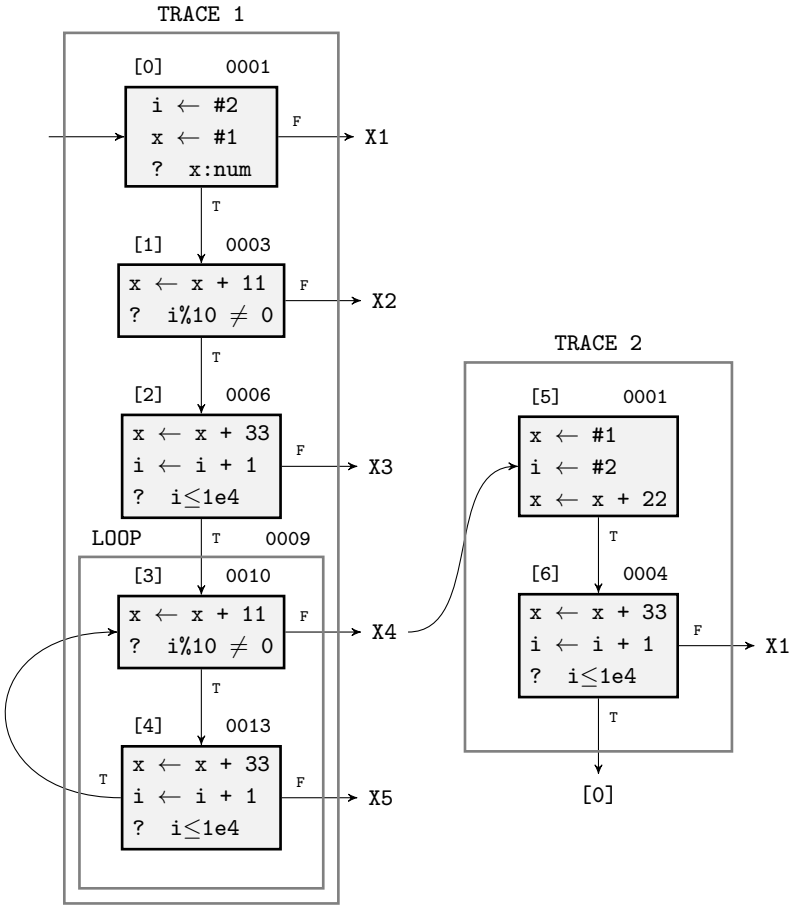

Figure 5.3: Trace flow diagram loop with if-statement

5.2.4 Nested loop

This section explains how the compiler generates traces in the case of nested loop. It creates a trace that refers to the inner loop and another trace for the outer loop.

```
1  -- Nested loop
2
3  local x = 0
4
5  for i=1,1e4,2 do     -- outer loop
6      x = x + 11
7      for j=2,1e3,4 do     -- inner loop
8          x = x + 22
9      end
10     x = x + 33
11 end
```

The instructions of the inner loop will be executed repeatedly at first. Thus, the inner loop becomes hot first and the compiler creates a trace (`TRACE 1`). At some point, also the outer loop becomes hot and the compiler generates another trace (`TRACE 2`) that is a side trace of the previous one.

Traces are organised in a reverse order if compared with the standard way of thinking the execution flow of nested loops. As it is shown in the bytecode below: `TRACE 1` (inner loop) contains the instruction $x = x + 22$; `TRACE 2` (outer loop) contains first the instruction $x = x + 33$ and then $x = x + 11$. Moreover, `TRACE 2` is a side trace that starts at the exit number 3 of `TRACE 1` (when the inner loop finished).

```
---- TRACE 1 start Ex.lua:7
0011  ADDVN    0   0   1  ; 22
0012  FORL     5 => 0011
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/3 Ex.lua:10
0013  ADDVN    0   0   2  ; 33
0014  FORL     1 => 0006
0006  ADDVN    0   0   0  ; 11
0007  KSHORT   5   2
0008  KSHORT   6 1000
0009  KSHORT   7   4
0010  JFORI    5 => 0013
---- TRACE 2 stop -> 1
```

The method of organising traces for nested loops is more clear when looking at the IR and the diagram.

```
---- TRACE 1 start Ex.lua:7
---- TRACE 1 IR
0001    int SLOAD  #6    CI
0002 >  num SLOAD  #1    T
0003  + num ADD    0002  +22
0004  + int ADD    0001  +4
0005 >  int LE     0004  +1000
0006 ------ LOOP ------------
0007  + num ADD    0003  +22
0008  + int ADD    0004  +4
0009 >  int LE     0008  +1000
0010    int PHI    0004  0008
0011    num PHI    0003  0007
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/3 Ex.lua:10
---- TRACE 2 IR
0001    num SLOAD  #1    PI
0002    num ADD    0001  +33
0003    num SLOAD  #2    I
0004    num ADD    0003  +2
0005 >  num LE     0004  +10000
0006    num ADD    0002  +11
---- TRACE 2 stop -> 1
```

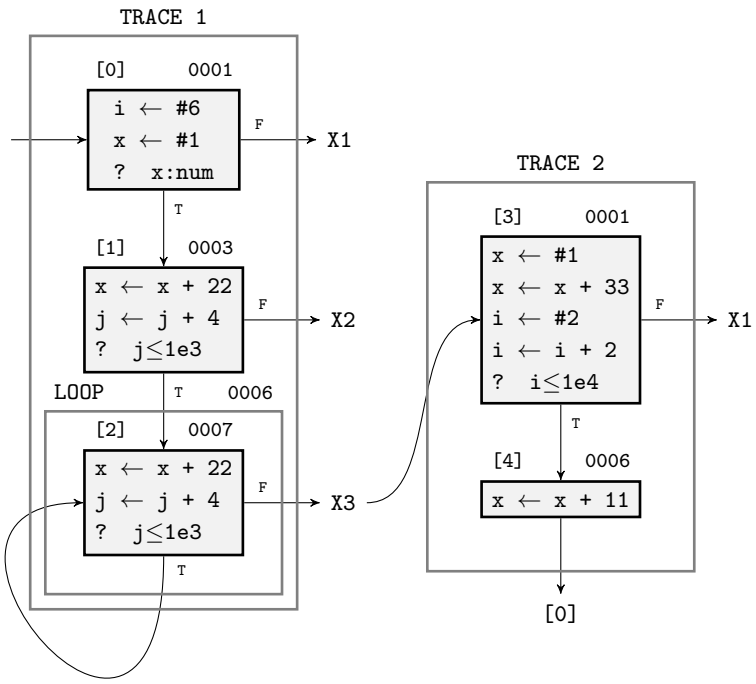

Figure 5.4: Trace flow diagram nested loop

The outer loop (`TRACE 2`) goes around the inner loop (`TRACE 1`) and joins it in block `[0]`.

On the other hand, if the inner loop had low iteration count, it would be unrolled and inlined[2].

[2]Conversation on the LuaJIT mailing list: `https://www.freelists.org/post/luajit/How-does-LuaJITs-trace-compiler-work,1`

5.3 Loop with two if-statements

In this section it will be explored how the JIT-compiler organises traces when there are two different if-statements within the same loop. The cases investigated are as follow: (i) the first if-statement condition becomes true (hot) before the second; (ii) the second becomes true (hot) before the first; (iii) the if-statement conditions become true (hot) at different time; (iv) the if-statement conditions become true (hot) at the same time.

5.3.1 Case 1

This example shows how traces are organised in a loop with 2 if-statements when the first condition becomes true (hot) before the second. In this case the truthfulness of the second if-condition implies the first.

```
1  -- Loop with 2 if-statement Example 1
2
3  local x = 0
4
5  for i=1,1e6 do
6    x = x + 11
7    if i%10 == 0 then    -- 1st if-statement
8      x = x + 22
9    end
10   x = x + 33
11   if i%20 == 0 then    -- 2nd if-statement
12     x = x + 44
13   end
14   x = x + 55
15 end
```

As it happened in the example 5.2.3, the execution flow skips most of the times the instructions contained in the if-statements and the compiler creates a trace (`TRACE 1`) with the instructions $x = x + 11$, $x = x + 33$, $x = x + 55$. By increasing i, the condition of the first if-statement becomes true more and more often, thus the compiler will generate a side trace (`TRACE 2`) that contains the instruction within the first if-statement $x = x + 22$ and what follows down to the loop "end". At some point, also the condition of the second if-statement becomes true repeatedly, thus the compiler creates a trace (`TRACE 3`) with the instruction within the second if-statement $x = x + 44$ and what follows down to the loop "end".

The bytecode below shows what was just explained.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   1  ; 11
0007  MODVN    5   4   2  ; 10
0008  ISNEN    5   3      ; 0
0009  JMP      5 => 0011
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   6  ; 20
0013  ISNEN    5   3      ; 0
```

```
0014  JMP       5 => 0016
0016  ADDVN     0   0   8  ; 55
0017  FORL      1 => 0006
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/5 Ex.lua:8
0010  ADDVN     0   0   4  ; 22
0011  ADDVN     0   0   5  ; 33
0012  MODVN     5   4   6  ; 20
0013  ISNEN     5   3      ; 0
```

```
0014  JMP       5 => 0016
0016  ADDVN     0   0   8  ; 55
0017  JFORL     1   1
---- TRACE 2 stop -> 1

---- TRACE 3 start 2/1 Ex.lua:12
0015  ADDVN     0   0   7  ; 44
0016  ADDVN     0   0   8  ; 55
0017  JFORL     1   1
---- TRACE 3 stop -> 1
```

Links between traces are more explicit in the IR and in the diagram.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001     int SLOAD  #2    CI
0002 >   num SLOAD  #1    T
0003     num ADD    0002  +11
0004     int MOD    0001  +10
0005 >   int NE     0004  +0
0006     num ADD    0003  +33
0007     int MOD    0001  +20
0008 >   int NE     0007  +0
0009   + num ADD    0006  +55
0010   + int ADD    0001  +1
0011 >   int LE     0010  +1000000
0012 ------ LOOP ------------
0013     num ADD    0009  +11
0014     int MOD    0010  +10
0015 >   int NE     0014  +0
0016     num ADD    0013  +33
0017     int MOD    0010  +20
0018 >   int NE     0017  +0
0019   + num ADD    0016  +55
0020   + int ADD    0010  +1
0021 >   int LE     0020  +1000000
0022     int PHI    0010  0020
0023     num PHI    0009  0019
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 start 1/5 Ex.lua:8
---- TRACE 2 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +22
0004     num ADD    0003  +33
0005     int MOD    0002  +20
0006 >   int NE     0005  +0
0007     num ADD    0004  +55
0008     int ADD    0002  +1
0009 >   int LE     0008  +1000000
0010     num CONV   0008  num.int
---- TRACE 2 stop -> 1

---- TRACE 3 start 2/1 Ex.lua:12
---- TRACE 3 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +44
0004     num ADD    0003  +55
0005     int ADD    0002  +1
0006 >   int LE     0005  +1000000
0007     num CONV   0005  num.int
---- TRACE 3 stop -> 1
```

Generally side traces are created when an exit is taken repeatedly. In this case `TRACE 2` starts at the exit number 5 of `TRACE 1` (line `0015`) and it joins `TRACE 1` at the end. `TRACE 3` starts at the exit number 1 of `TRACE 2` (line `0006`) and it joins `TRACE 1` at the end. Thus, `TRACE 2` is a side trace of `TRACE 1` and `TRACE 3` is a side trace of `TRACE 2`, both joining `TRACE 1` at their ends.

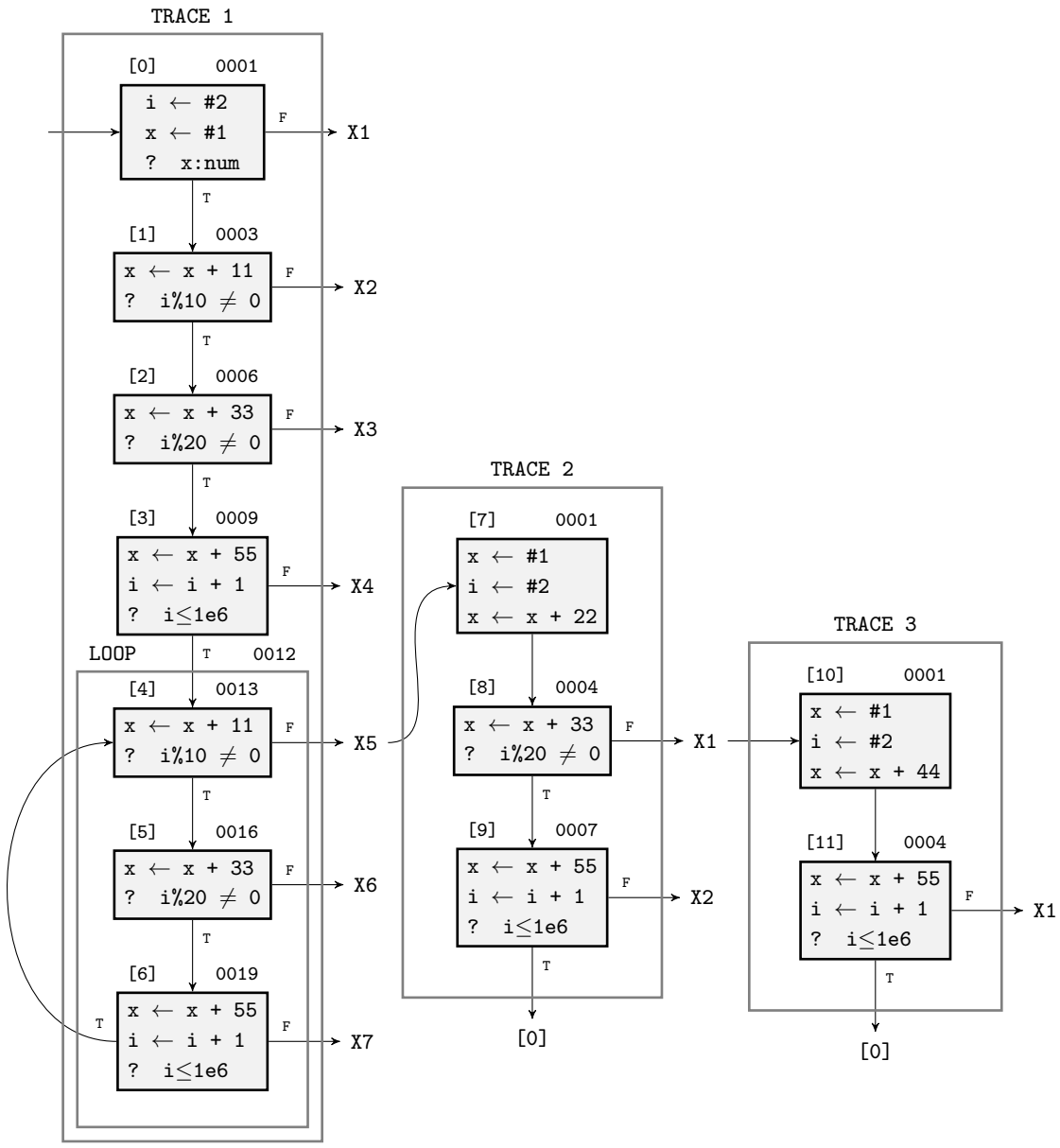

Figure 5.5: Trace flow diagram loop with 2 if-statements Example 1

5.3.2 Case 2

This example is the same as the previous one, but the order of the two if-statement is reversed. In particular, it investigates how traces are organised in a loop with two if-statements when the second condition becomes true (hot) before the first. In this case the truthfulness of the first if-condition implies the second.

Even if the change from the previous example is small, it causes a big difference in the traces organisation.

```
1  -- Loop with 2 if-statement Example 2
2
3  local x = 0
4
5  for i=1,1e6 do
6    x = x + 11
7    if i%20 == 0 then    -- 1st if-statement
8      x = x + 22
9    end
10   x = x + 33
11   if i%10 == 0 then    -- 2nd if-statement
12     x = x + 44
13   end
14   x = x + 55
15 end
```

The compiler creates the first trace (`TRACE 1`) with the same logic of the previous example. It contains the instructions $x = x + 11$, $x = x + 33$, $x = x + 55$. By increasing i, the condition of the second if-statement becomes true more and more often, thus the compiler will generate a side trace (`TRACE 3`) that contains the instruction within the second if-statement $x = x + 44$ and what follows. At some point, also the condition of the first if-statement becomes true repeatedly, thus the compiler creates a trace (`TRACE 2`) that contains instructions within both the if-statements and what follows.

The bytecode below shows what was just explained.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   1  ; 11
0007  MODVN    5   4   2  ; 20
0008  ISNEN    5   3      ; 0
0009  JMP      5 => 0011
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   6  ; 10
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0016  ADDVN    0   0   8  ; 55
0017  FORL     1 => 0006
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/5 Ex.lua:8
0010  ADDVN    0   0   4  ; 22
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   6  ; 10
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0015  ADDVN    0   0   7  ; 44
0016  ADDVN    0   0   8  ; 55
0017  JFORL    1   1
---- TRACE 2 stop -> 1

---- TRACE 3 start 1/6 Ex.lua:12
0015  ADDVN    0   0   7  ; 44
0016  ADDVN    0   0   8  ; 55
0017  JFORL    1   1
---- TRACE 3 stop -> 1
```

The IR displayed below shows that `TRACE 2` starts at the exit number 5 of `TRACE 1` (line `0015`) and it joins `TRACE 1` at the end. `TRACE 3` starts at the exit number 6 of `TRACE 1` (line `0018`) and it joins `TRACE 1` at the end. Thus, both `TRACE 2` and `TRACE 3` are side traces of `TRACE 1`.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001     int SLOAD  #2    CI
0002 >   num SLOAD  #1    T
0003     num ADD    0002  +11
0004     int MOD    0001  +20
0005 >   int NE     0004  +0
0006     num ADD    0003  +33
0007     int MOD    0001  +10
0008 >   int NE     0007  +0
0009   + num ADD    0006  +55
0010   + int ADD    0001  +1
0011 >   int LE     0010  +1000000
0012 ------ LOOP ------------
0013     num ADD    0009  +11
0014     int MOD    0010  +20
0015 >   int NE     0014  +0
0016     num ADD    0013  +33
0017     int MOD    0010  +10
0018 >   int NE     0017  +0
0019   + num ADD    0016  +55
0020   + int ADD    0010  +1
0021 >   int LE     0020  +1000000
0022     int PHI    0010  0020
0023     num PHI    0009  0019
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 start 1/5 Ex.lua:8
---- TRACE 2 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +22
0004     num ADD    0003  +33
0005     int MOD    0002  +10
0006 >   int EQ     0005  +0
0007     num ADD    0004  +44
0008     num ADD    0007  +55
0009     int ADD    0002  +1
0010 >   int LE     0009  +1000000
0011     num CONV   0009  num.int
---- TRACE 2 stop -> 1
```

```
---- TRACE 3 start 1/6 Ex.lua:12
---- TRACE 3 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +44
0004     num ADD    0003  +55
0005     int ADD    0002  +1
0006 >   int LE     0005  +1000000
0007     num CONV   0005  num.int
---- TRACE 3 stop -> 1
```

It is possible to make a comparison between this example and the previous one.

In the example of case 1, LuaJIT creates a longer chain of traces attached one to another in a sequence of sidetraces. The root trace (`TRACE 1`), which covers the most critical hotpath, contains only one guard that is actually failing when the if-conditions are true.

On the other hand, in the example of case 2 all the sidetraces are attached to the root trace (`TRACE 1`), hence there is no chain. The root trace, which covers the most critical hotpath, contains both the two guards that are actually failing.

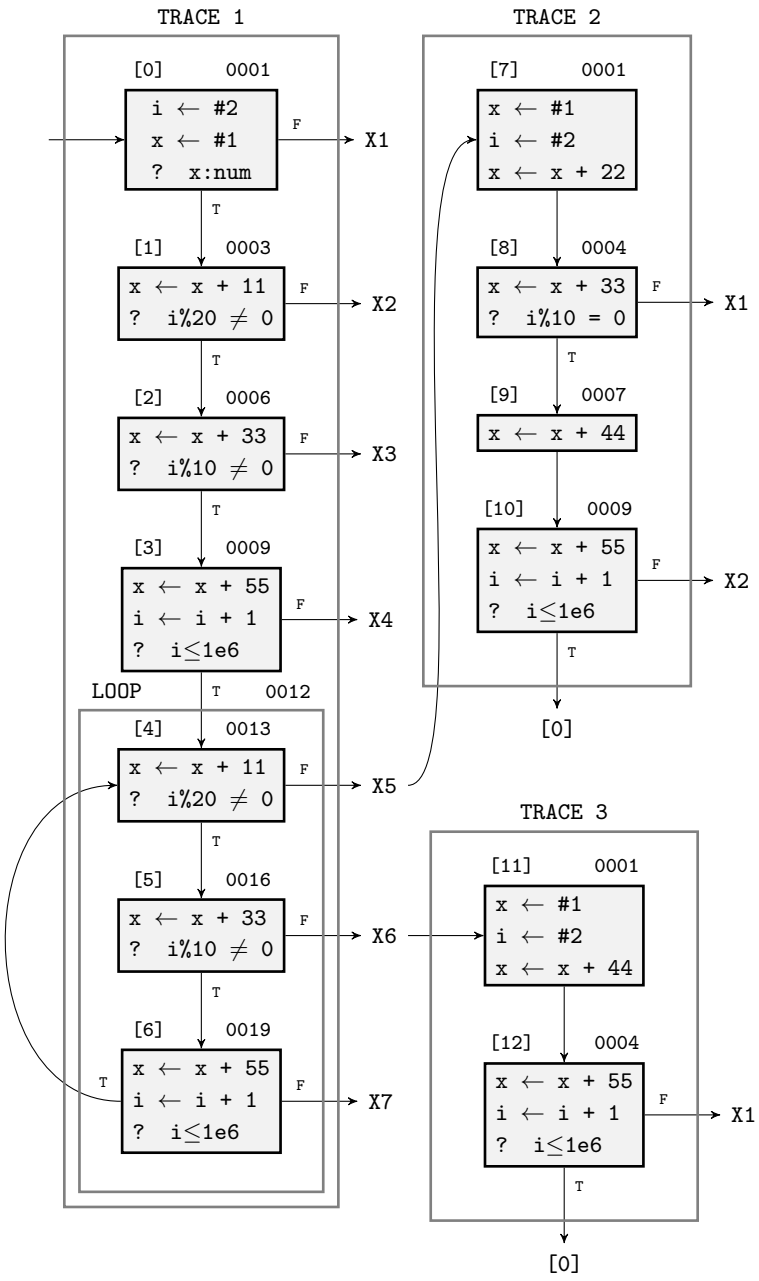

Figure 5.6: Trace flow diagram loop with 2 if-statements Example 2

5.3.3 Case 3

This example shows how traces are organised in a loop with two if-statements when both conditions become true (hot) at the same time. The compiler does not generate a side trace for each if-statement, but it creates only a unique side trace. In this case the truthfulness of the first if-condition implies the second and vice versa.

```
1  -- Loop with 2 if-statement  Example 3
2
3  local x = 0
4
5  for i=1,1e6 do
6    x = x + 11
7    if i%10 == 0 then    -- 1st if-statement
8      x = x + 22
9    end
10   x = x + 33
11   if i%10 == 0 then    -- 2nd if-statement
12     x = x + 44
13   end
14   x = x + 55
15 end
```

The compiler produces the first trace (`TRACE 1`) with the same logic of the previous examples. It contains the instructions $x = x + 11$, $x = x + 33$, $x = x + 55$. By increasing i, the condition of both if-statements becomes true at the same time more and more often. Thus, the compiler will generate a side trace (`TRACE 2`) that contains the instruction with both if-statements and what follows.

The bytecode below shows what was just explained.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   1  ; 11
0007  MODVN    5   4   2  ; 10
0008  ISNEN    5   3      ; 0
0009  JMP      5 => 0011
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   2  ; 10
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0016  ADDVN    0   0   7  ; 55
0017  FORL     1 => 0006
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 start 1/4 Ex.lua:8
0010  ADDVN    0   0   4  ; 22
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   2  ; 10
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0015  ADDVN    0   0   6  ; 44
0016  ADDVN    0   0   7  ; 55
0017  JFORL    1   1
---- TRACE 2 stop -> 1
```

In the IR below it is shown that `TRACE 2` starts at the exit number 4 of `TRACE 1` (line `0013`) and it joins `TRACE 1` at the end. The exit of `TRACE 2` at `line 0006` will never be taken because $i\%10 = 0$ will always be true, since this was the condition that led the execution flow to enter in the side trace itself (see Diagram 5.7).

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001     int SLOAD  #2    CI
0002 >   num SLOAD  #1    T
0003     num ADD    0002  +11
0004     int MOD    0001  +10
0005 >   int NE     0004  +0
0006     num ADD    0003  +33
0007  +  num ADD    0006  +55
0008  +  int ADD    0001  +1
0009 >   int LE     0008  +1000000
0010 ------ LOOP ------------
0011     num ADD    0007  +11
0012     int MOD    0008  +10
0013 >   int NE     0012  +0
0014     num ADD    0011  +33
0015  +  num ADD    0014  +55
0016  +  int ADD    0008  +1
0017 >   int LE     0016  +1000000
0018     int PHI    0008  0016
0019     num PHI    0007  0015
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/4 Ex.lua:8
---- TRACE 2 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +22
0004     num ADD    0003  +33
0005     int MOD    0002  +10
0006 >   int EQ     0005  +0
0007     num ADD    0004  +44
0008     num ADD    0007  +55
0009     int ADD    0002  +1
0010 >   int LE     0009  +1000000
0011     num CONV   0009  num.int
---- TRACE 2 stop -> 1
```

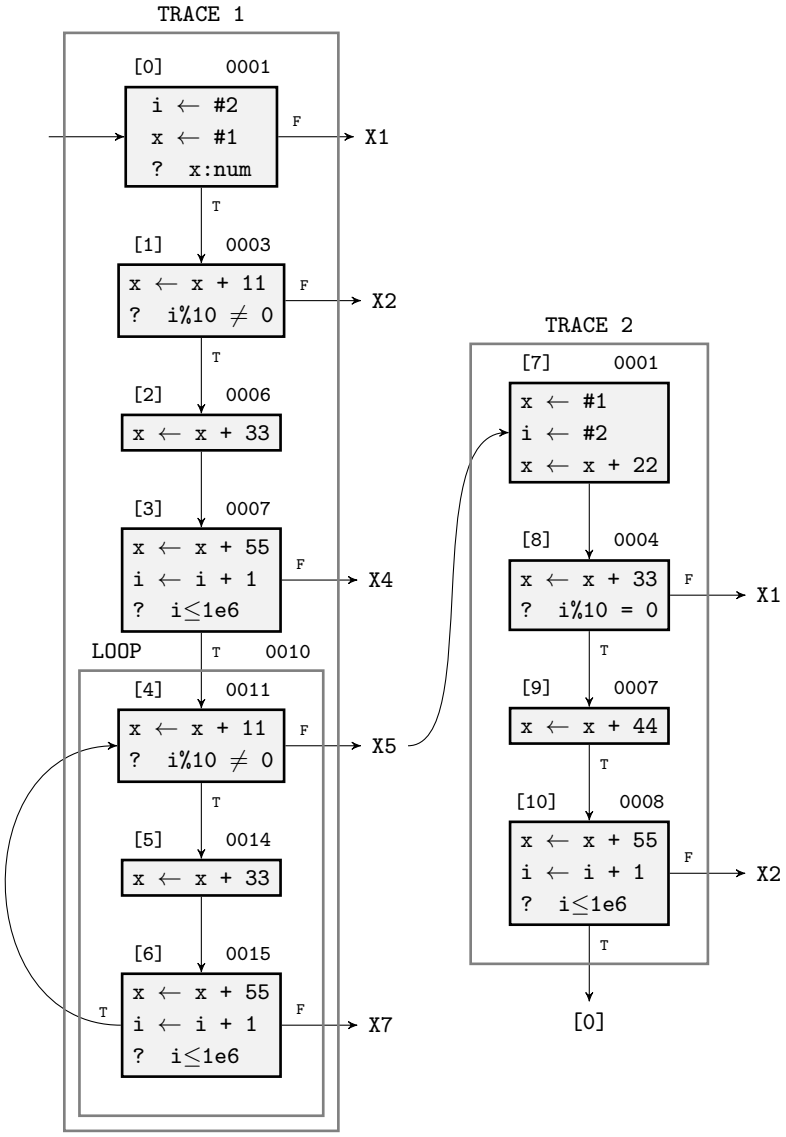

Figure 5.7: Trace flow diagram loop with 2 if-statements Example 3

5.3.4 Case 4

This example shows how traces are organised in a loop with two if-statements when the first condition becomes hot before the second, but the conditions become true mostly at different time. In this case the truthfulness of the second implies the first just for some iteration of the loop (e.g. $i = 60, 120, ...$).

```
1  -- Loop with 2 if-statement Example 4
2
3  local x = 0
4
5  for i=1,1e6 do
6    x = x + 11
7    if i%15 == 0 then    -- 1st if-statement
8      x = x + 22
9    end
10   x = x + 33
11   if i%20 == 0 then    -- 2nd if-statement
12     x = x + 44
13   end
14   x = x + 55
15 end
```

The compiler creates the first trace (`TRACE 1`) with the same logic of the previous examples. It contains the instructions $x = x + 11$, $x = x + 33$, $x = x + 55$. By increasing i, the condition of the first if-statement becomes true more and more often, thus the compiler will generate a side trace (`TRACE 2`) that contains the instruction within the first if-statement $x = x + 22$ and what follows. At some point, also the condition of the second if-statement becomes true repeatedly, thus the compiler creates a trace (`TRACE 3`) with the instruction within the second if-statement $x = x + 44$ and what follows. Both `TRACE 2` and `TRACE 3` are side trace of `TRACE 1`.

Later on, when both the conditions becomes true at the same time repeatedly, the compiler generates another trace (`TRACE 4`) that contains the instruction within the second if-statement $x = x + 44$ and what follows. `TRACE 4` starts as a side trace of `TRACE 2`.

The bytecode below shows what was just explained.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   1  ; 11
0007  MODVN    5   4   2  ; 15
0008  ISNEN    5   3      ; 0
0009  JMP      5 => 0011
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   6  ; 20
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0016  ADDVN    0   0   8  ; 55
0017  FORL     1 => 0006
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 start 1/5 Ex.lua:8
0010  ADDVN    0   0   4  ; 22
0011  ADDVN    0   0   5  ; 33
0012  MODVN    5   4   6  ; 20
0013  ISNEN    5   3      ; 0
0014  JMP      5 => 0016
0016  ADDVN    0   0   8  ; 55
0017  JFORL    1   1
---- TRACE 2 stop -> 1

---- TRACE 3 start 1/6 Ex.lua:12
```

```
0015  ADDVN    0   0   7  ; 44
0016  ADDVN    0   0   8  ; 55
0017  JFORL    1   1
---- TRACE 3 stop -> 1
```

```
---- TRACE 4 start 2/1 Ex.lua:12
0015  ADDVN    0   0   7  ; 44
0016  ADDVN    0   0   8  ; 55
0017  JFORL    1   1
---- TRACE 4 stop -> 1
```

The IR displayed below shows that `TRACE 2` starts at the exit number 5 of `TRACE 1` (line `0015`) and it joins `TRACE 1` at the end. `TRACE 3` starts at the exit number 6 of `TRACE 1` (line `0018`) and it joins `TRACE 1` at the end. `TRACE 4` starts at the exit number 1 of `TRACE 2` (line `0006`) and it joins `TRACE 1` at the end.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
0001     int SLOAD  #2    CI
0002 >   num SLOAD  #1    T
0003     num ADD    0002  +11
0004     int MOD    0001  +15
0005 >   int NE     0004  +0
0006     num ADD    0003  +33
0007     int MOD    0001  +20
0008 >   int NE     0007  +0
0009   + num ADD    0006  +55
0010   + int ADD    0001  +1
0011 >   int LE     0010  +1000000
0012 ------ LOOP ------------
0013     num ADD    0009  +11
0014     int MOD    0010  +15
0015 >   int NE     0014  +0
0016     num ADD    0013  +33
0017     int MOD    0010  +20
0018 >   int NE     0017  +0
0019   + num ADD    0016  +55
0020   + int ADD    0010  +1
0021 >   int LE     0020  +1000000
0022     int PHI    0010  0020
0023     num PHI    0009  0019
---- TRACE 1 stop -> loop

---- TRACE 2 start 1/5 Ex.lua:8
---- TRACE 2 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +22
0004     num ADD    0003  +33
0005     int MOD    0002  +20
0006 >   int NE     0005  +0
0007     num ADD    0004  +55
0008     int ADD    0002  +1
0009 >   int LE     0008  +1000000
0010     num CONV   0008  num.int
---- TRACE 2 stop -> 1

---- TRACE 3 start 1/6 Ex.lua:12
---- TRACE 3 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +44
0004     num ADD    0003  +55
0005     int ADD    0002  +1
0006 >   int LE     0005  +1000000
0007     num CONV   0005  num.int
---- TRACE 3 stop -> 1

---- TRACE 4 start 2/1 Ex.lua:12
---- TRACE 4 IR
0001     num SLOAD  #1    PI
0002     int SLOAD  #2    PI
0003     num ADD    0001  +44
0004     num ADD    0003  +55
0005     int ADD    0002  +1
0006 >   int LE     0005  +1000000
0007     num CONV   0005  num.int
---- TRACE 4 stop -> 1
```

Possible trace paths covered by the execution flow are: (i) `TRACE 1`: if both conditions are false; (ii) `TRACE 1`, `TRACE 2`: if the first condition is true and the second is false; (iii) `TRACE 1`, `TRACE 3`: if the first condition is false and the second is true; (iv) `TRACE 1`, `TRACE 2`, `TRACE 4`: if both conditions are true.

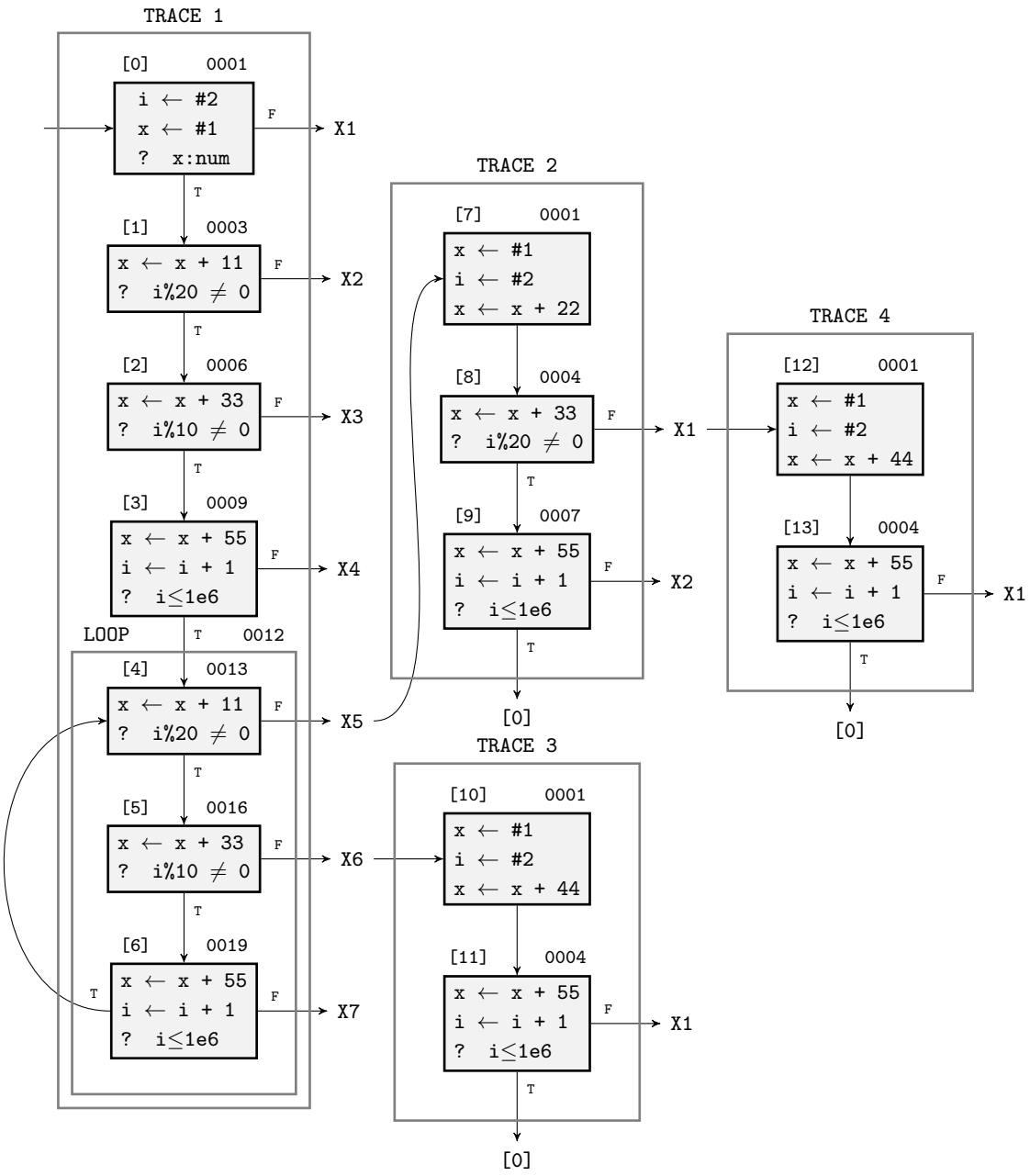

Figure 5.8: Trace flow diagram loop with 2 if-statements Example 4

5.4 Nested loop with more inner loops

The goal of this example is to explain how the compiler organises traces in the case of nested loop with two (or more) inner loops. The compiler generates:

a trace for each inner loop, a trace (or more) to connect them and a trace for the outer loop that goes around the inner loops.

```
1  -- Nested loop with 2 inner loops
2
3  local x = 0
4
5  for i=1,1e4,2 do        -- outer loop
6    x = x + 11
7    for j=3,2e4,4 do    -- inner loop (LOOP 1)
8      x = x + 22
9    end
10   x = x + 33
11   for k=5,3e4,6 do    -- inner loop (LOOP 2)
12     x = x + 44
13   end
14   x = x + 55
15 end
```

In this case the instructions of the inner loops will be executed repeatedly at first. Thus, the inner loops become hot first and the compiler creates a trace for each of them (`TRACE 1`, `TRACE 2`). At some point, also the outer loop becomes hot and the compiler generates a trace (`TRACE 3`) to connect the two inner loops (with the instruction $x = x+33$) and a trace (`TRACE 4`) that goes around the inner loops (with the instructions $x = x+55$, $x = x+11$). Note that `TRACE 1`, `TRACE 2` are root traces and `TRACE 3`, `TRACE 4` are sidetraces.

The bytecode below shows what was just explained.

```
---- TRACE 1 start *.lua:7
0011  ADDVN    0   0   1  ; 22
0012  FORL     5 => 0011
---- TRACE 1 stop -> loop

---- TRACE 2 start *.lua:11
0018  ADDVN    0   0   3  ; 44
0019  FORL     5 => 0018
---- TRACE 2 stop -> loop

---- TRACE 3 start 1/3 *.lua:10
0013  ADDVN    0   0   2  ; 33
0014  KSHORT   5   5
0015  KSHORT   6 30000
0016  KSHORT   7   6
0017  JFORI    5 => 0020
---- TRACE 3 stop -> 2

---- TRACE 4 start 2/3 *.lua:14
0020  ADDVN    0   0   4  ; 55
0021  FORL     1 => 0006
0006  ADDVN    0   0   0  ; 11
0007  KSHORT   5   3
0008  KSHORT   6 20000
0009  KSHORT   7   4
0010  JFORI    5 => 0013
---- TRACE 4 stop -> 1
```

The IR shows the details of the traces organisation. `TRACE 1` and `TRACE 2` are independent traces. `TRACE 3` starts at the exit number 3 of `TRACE 1` (line `0009`) and it joins `TRACE 2` at the end (this is the connection of the two inner loops). `TRACE 4` starts at the exit number 3 of `TRACE 2` (line `0009`) and it joins `TRACE 1` at the end. This is the part of the outer loop that goes around the inner loops.

```
---- TRACE 1 start *.lua:7
---- TRACE 1 IR
0001     int SLOAD  #6    CI
0002 >   num SLOAD  #1    T
0003   + num ADD    0002  +22
0004   + int ADD    0001  +4
0005 >   int LE     0004  +20000
0006 ------ LOOP ------------
0007   + num ADD    0003  +22
0008   + int ADD    0004  +4
0009 >   int LE     0008  +20000
0010     int PHI    0004  0008
0011     num PHI    0003  0007
---- TRACE 1 stop -> loop

---- TRACE 2 start *.lua:11
---- TRACE 2 IR
0001     int SLOAD  #6    CI
0002 >   num SLOAD  #1    T
0003   + num ADD    0002  +44
0004   + int ADD    0001  +6
0005 >   int LE     0004  +30000
0006 ------ LOOP ------------
0007   + num ADD    0003  +44
0008   + int ADD    0004  +6
0009 >   int LE     0008  +30000
0010     int PHI    0004  0008
0011     num PHI    0003  0007
---- TRACE 2 stop -> loop

---- TRACE 3 start 1/3 *.lua:10
---- TRACE 3 IR
0001     num SLOAD  #1    PI
0002     num ADD    0001  +33
---- TRACE 3 stop -> 2

---- TRACE 4 start 2/3 *.lua:14
---- TRACE 4 IR
0001     num SLOAD  #1    PI
0002     num ADD    0001  +55
0003     num SLOAD  #2    I
0004     num ADD    0003  +2
0005 >   num LE     0004  +10000
0006     num ADD    0002  +11
---- TRACE 4 stop -> 1
```

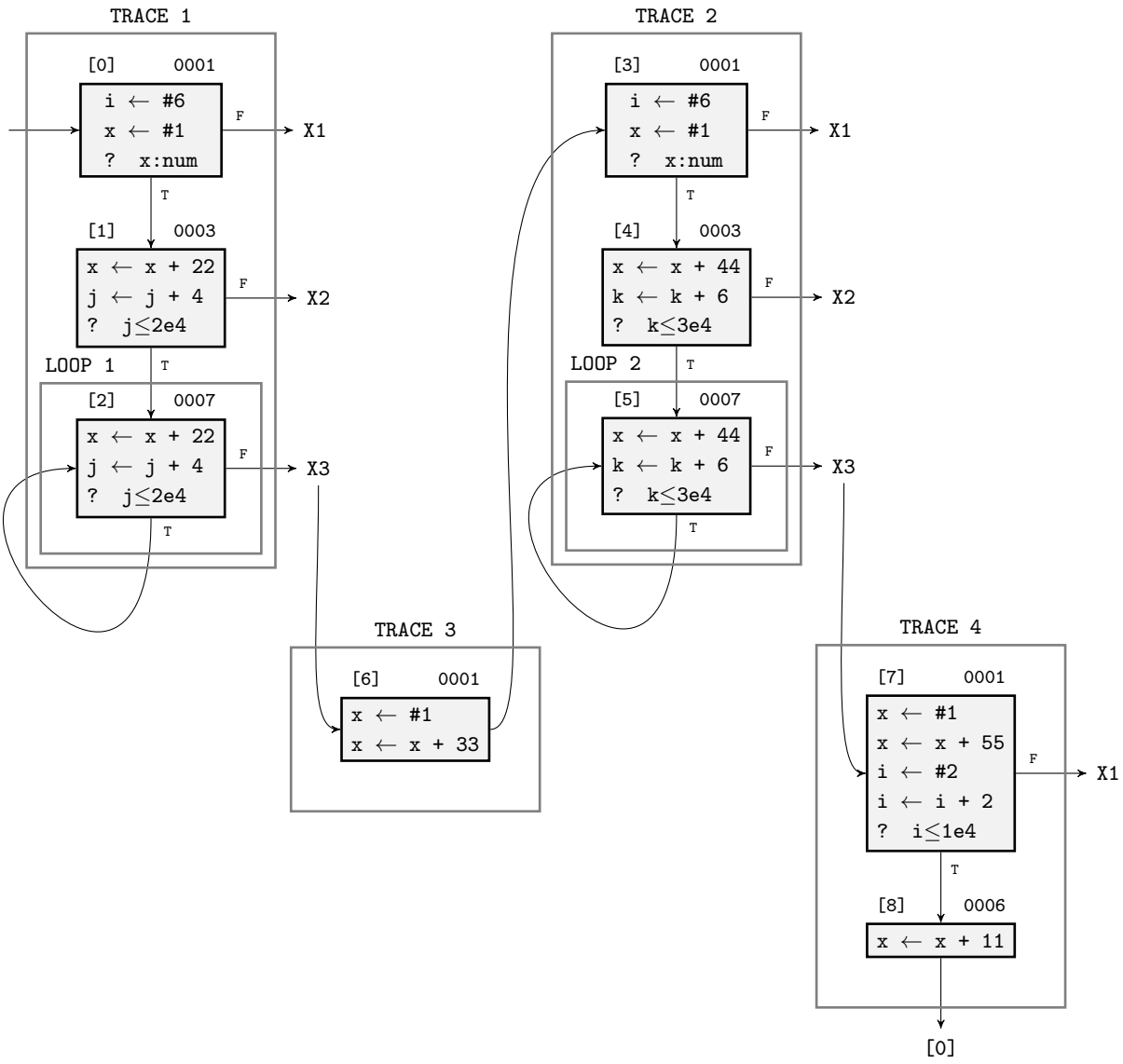

Figure 5.9: Trace flow diagram loop with 2 inner loops

The same structure is maintained when the number of inner loops increases. If n is the number of inner loops, the compiler generates: n traces for the n inner loops, $n-1$ traces to connect the inner loops and a final trace for the outer loop that goes around the inner loops.

The code below describes the case of $n = 3$.

```
1 -- Nested loop with 3 inner loops
2
3 local x = 0
```

```
4
5  for i=1,1e4 do           -- outer loop
6    x = x + 11
7      for j=1,2e4,2 do     -- inner loop (LOOP 1)
8      x = x + 22
9    end
10   x = x + 33
11     for k=1,3e4,3 do     -- inner loop (LOOP 2)
12     x = x + 44
13   end
14   x = x + 55
15     for t=1,4e4,4 do     -- inner loop (LOOP 3)
16     x = x + 66
17   end
18   x = x + 77
19 end
```

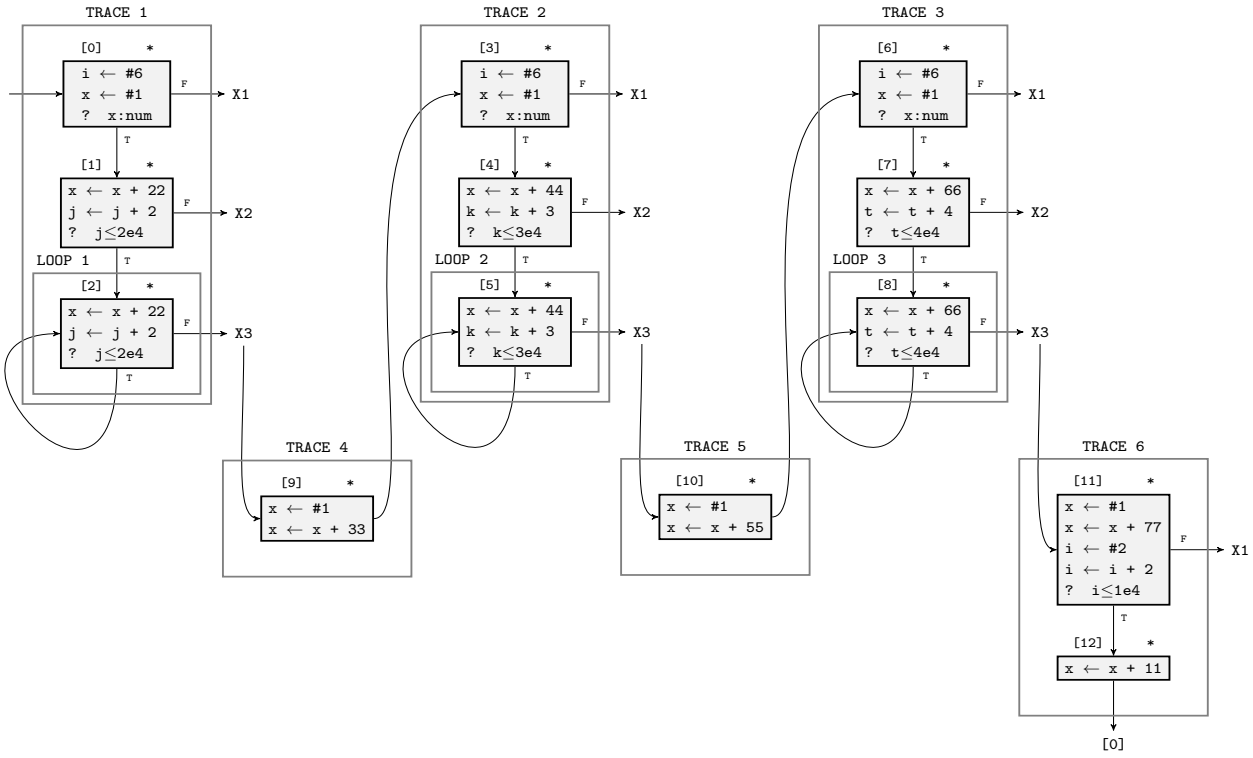

Figure 5.10: Trace flow diagram loop with 3 inner loops

5.5 Recursive functions

In this section it will be shown what is the behaviour of the compiler when dealing with recursive functions. In particular, both non-tail recursive and tail recursive functions have been investigated.

5.5.1 Non-tail recursive function

This example consists in a non-tail recursive factorial function. The compiler does not create a loop structure because the function is non-tail recursive.

```
1 -- Non-tail recursive factorial
2
3 local function factorial(n)
4   if n > 0 then
5     return n * factorial(n-1)
6   end
7   return 1
8 end
```

The function calls itself as a standard recursive function because an operation occurs on the call result before returning. Since it is non-tail recursive, when making a recursive call, the return address needs to be pushed onto the call stack then jump to the called function. This means that it needs a call stack whose size is linear with the depth of the recursive calls.

In the bytecode below the depth of the recursive calls is represented by dots in the second column. In this case the compiler unrolls three recursive calls. Thus, the trace created contains three function calls with the instruction `FUNCF` (lines `0000`) and it ends with an up-recursion `-- TRACE 1 stop -> up-recursion`.

```
---- TRACE 1 start Ex.lua:3
0001  KSHORT   1   0
0002  ISGE     1   0
0003  JMP      1 => 0009
0004  UGET     1   0; factorial
0005  SUBVN    2   0   0  ; 1
0006  CALL     1   2   2
0000  . FUNCF    3  ; Ex.lua:3
0001  . KSHORT   1   0
0002  . ISGE     1   0
0003  . JMP      1 => 0009
0004  . UGET     1   0 ; factorial
0005  . SUBVN    2   0   0  ; 1
0006  . CALL     1   2   2
0000  . . FUNCF    3 ; Ex.lua:3
0001  . . KSHORT   1   0
0002  . . ISGE     1   0
0003  . . JMP      1 => 0009
0004  . . UGET     1   0 ; factorial
0005  . . SUBVN    2   0   0  ; 1
0006  . . CALL     1   2   2
0000  . . . FUNCF    3 ; Ex.lua:3
---- TRACE 1 stop -> up-recursion
```

The IR generated is the following.

```
---- TRACE 1 start Ex.lua:3
---- TRACE 1 IR
0001 >  num SLOAD  #1    T
0002 >  num GT     0001  +0
0003    fun SLOAD  #0    R
0004 >  fun EQ     0003  Ex.lua:3
0005    num SUB    0001  +1
0006 >  num GT     0005  +0
0007    num SUB    0005  +1
0008 >  num GT     0007  +0
0009    num SUB    0007  +1
---- TRACE 1 stop -> up-recursion
```

5.5.2 Tail recursive function

This example consists in a tail recursive factorial function. In this case, the compiler create a loop structure because the function is tail recursive.

```
1 -- Tail recursive factorial
2
3 local function factorial(n, r)
4   r = r or 1
5   if n > 0 then
6     return factorial(n-1, n*r)
7   end
8   return r
9 end
```

Since the function is tail recursive as soon as there is a return from the recursive call the execution flows goes immediately to a return as well. It skips the entire chain of recursive functions returning and it returns straight to the original caller. There is no need of a call stack for the recursive calls.

As a matter of fact, there are no dots in the bytecode below because there is no depth of recursive calls. Also in this case the compiler unrolls three recursive calls. Thus, the trace created contains three function calls with the instruction `FUNCF` (lines `0000`) and it ends with a tail recursion `-- TRACE 1 stop -> tail-recursion`.

```
---- TRACE 1 start Ex.lua:3
0001  IST          1
0002  JMP      2 => 0004
0004  KSHORT   2   0
0005  ISGE     2   0
0006  JMP      2 => 0011
0007  UGET     2   0; factorial
0008  SUBVN    3   0   0  ; 1
0009  MULVV    4   0   1
0010  CALLT    2   3
0000  FUNCF    5 ; Ex.lua:3
0001  IST          1
0002  JMP      2 => 0004
0004  KSHORT   2   0
0005  ISGE     2   0
0006  JMP      2 => 0011
```

```
0007  UGET     2   0 ; factorial
0008  SUBVN    3   0   0  ; 1
0009  MULVV    4   0   1
0010  CALLT    2   3
0000  FUNCF    5 ; Ex.lua:3
0001  IST          1
0002  JMP      2 => 0004
0004  KSHORT   2   0
0005  ISGE     2   0
0006  JMP      2 => 0011
0007  UGET     2   0 ; factorial
0008  SUBVN    3   0   0  ; 1
0009  MULVV    4   0   1
0010  CALLT    2   3
0000  FUNCF    5 ; Ex.lua:3
---- TRACE 1 stop -> tail-recursion
```

The fact that the compiler transforms tail-recursive functions in loops is more explicit in the IR (see the `-- LOOP --` label at line `0014`). This is a standard compiler transformation.

```
---- TRACE 1 start Ex.lua:3
---- TRACE 1 IR
0001 >  num SLOAD  #2    T
0002 >  num SLOAD  #1    T
0003 >  num GT     0002  +0
0004    fun SLOAD  #0    R
0005 >  fun EQ     0004  Ex.lua:3
0006    num SUB    0002  +1
0007    num MUL    0002  0001
0008 >  num GT     0006  +0
0009    num SUB    0006  +1
0010    num MUL    0007  0006
0011 >  num GT     0009  +0
0012  + num SUB    0009  +1
0013  + num MUL    0010  0009
0014 ------ LOOP ------------
0015 >  num GT     0012  +0
0016    num SUB    0012  +1
0017    num MUL    0013  0012
0018 >  num GT     0016  +0
0019    num SUB    0016  +1
0020    num MUL    0017  0016
0021 >  num GT     0019  +0
0022  + num SUB    0019  +1
0023  + num MUL    0020  0019
0024    num PHI    0013  0023
0025    num PHI    0012  0022
---- TRACE 1 stop -> tail-recursion
```

5.6 Stitch trace

Trace stitching is a feature introduced in LuaJIT version 2.1 which allows traces to stop at a classic C function or a not-compiled built-in, return to the interpreter, run the C function or built-in and then start a new trace after it returns. This is not particularly efficient, but it avoids trace aborts due to NYI functions, which would previously force the whole code path around such a function to be interpreted.

Trace stitching can be seen as the equivalent of disabling the JIT (`jit.off()`) before calling a C function or built-in, then enabling the JIT (`jit.on()`) just after the function call.

The bytecode instructions in which trace stitching feature plays a role are the following:

Bytecode ins	Description
FUNCC	Call C function via classic API
FUNCCW	Call wrapped C function via classic API

Table 5.6: Bytecode instructions for stitch trace

Here we propose a concrete example which describes trace stitching.

```lua
-- Stitch trace

local x = 0

for i=1,100 do
  x = x + 11
  os.clock()
  x = x + 22
end
```

The bytecode instructions produced by LuaJIT are the following.

```
---- TRACE 1 start Ex.lua:5
0006  ADDVN    0   0   0  ; 11
0007  GGET     5   0      ; "os"
0008  TGETS    5   5   1  ; "clock"
0009  CALL     5   1   1
0000  . FUNCC              ; os.clock
---- TRACE 1 stop -> stitch
```

```
---- TRACE 2 start 1/stitch Ex.lua:7
0010  ADDVN    0   0   1  ; 22
0011  JFORL    1   1
---- TRACE 2 stop -> 1
```

The instruction `x=x+11` is computed into `TRACE 1`, then `os.clock()` is run by the interpreter, and finally `x=x+22` is computed into `TRACE 2`.

The IR respects the same logic: `TRACE 1` is a root trace which is linked to trace stitching at its end (`---- TRACE 1 stop -> stitch`); `TRACE 2` is a stitch trace connected to `TRACE 1` (`---- TRACE 2 start 1/stitch Ex.lua:7`, the number `1` refers to `TRACE 1`). Finally `TRACE 2` loops back to `TRACE 1` because we are considering a `for` loop.

```
---- TRACE 1 start Ex.lua:5
---- TRACE 1 IR
....         SNAP   #0
0001    int  SLOAD  #2    CI
0002 >  num  SLOAD  #1    T
0003    num  ADD    0002  +11
0004    fun  SLOAD  #0    R
0005    tab  FLOAD  0004  func.env
0006    int  FLOAD  0005  tab.hmask
0007 >  int  EQ     0006  +63
0008    p32  FLOAD  0005  tab.node
0009 >  p32  HREFK  0008  "os" @56
0010 >  tab  HLOAD  0009
0011    int  FLOAD  0010  tab.hmask
0012 >  int  EQ     0011  +15
0013    p32  FLOAD  0010  tab.node
0014 >  p32  HREFK  0013  "clock" @14
0015 >  fun  HLOAD  0014
0016 >  fun  EQ     0015  os.clock
0017    num  CONV   0001  num.int
....         SNAP   #1
---- TRACE 1 stop -> stitch
```

```
---- TRACE 2 start 1/stitch Ex.lua:7
---- TRACE 2 IR
....         SNAP   #0
0001 >  num  SLOAD  #1    T
0002    num  ADD    0001  +22
0003    num  SLOAD  #2    I
0004    num  ADD    0003  +1
....         SNAP   #1
0005 >  num  LE     0004  +100
....         SNAP   #2
---- TRACE 2 stop -> 1
```

Chapter 6

Analysis tools

This chapter presents some diagnostic tools for the analysis of LuaJIT. The compiler framework provides a set of tools that helps to investigate what happens under the hood while executing a program with LuaJIT. In particular we will illustrate: (i) the verbose mode, (ii) the profiler, and (iii) the dump mode.

Over the years, the LuaJIT community realised that a key aspect to understand LuaJIT behaviour more in details and to facilitate its usage would be to improve the reported information provided by the LuaJIT "standard" tools. In these regards, we implemented a small extension of the dump mode in order to get more specific information on traces. The goal is to provide to the user some insights for writing JIT-friendly code, since in some specific circumstances certain patterns of code can be preferred to others.

Another interesting tool for interactive software diagnostics, called Studio [48], was born in the context of RaptorJIT [20], a fork of LuaJIT. It is an interactive tools for inspecting and cross-referencing trace and profiler data.

6.1 Verbose mode

This module shows verbose information about the progress of the JIT compiler, printing a line for each generated trace. It is useful to inspect which code has been compiled or where the compiler stops and falls back to the interpreter.

Some examples of its usage are shown below (note that when indicating the file name, the file is overwritten every time the module is started).

```
luajit -jv -e "for i=1,1000 do for j=1,1000 do end end"
luajit -jv=myapp.out myapp.lua
```

The output of the second example could be like this:

```
[TRACE   1 myapp.lua:1 loop]
[TRACE   2 (1/3) myapp.lua:1 -> 1]
```

The first number in each line indicates the internal trace number. Then, it prints the file name "myapp.lua" and the line number ":1" where the trace started. Sidetraces also show in parentheses "(1/3)" the parent trace number and the exit number from where they are attached. An arrow at the end shows where the trace links to "$->$ 1" unless it loops to itself.

When a trace aborts the output is the following:

```
[TRACE --- foo.lua:44 -- leaving loop in root trace at foo:lua:50]
```

Trace aborts are quite common, even in programs which can be fully compiled. The compiler may retry several times until it finds a suitable trace.

6.2 Profiler

This module is a command line interface to the built-in low-overhead profiler of LuaJIT. The lower-level API of the profiler is accessible via the "jit.profile" module or the LuaJIT_profile_* C API.

Some examples of using this mode are shown below.

```
luajit -jp myapp.lua
luajit -jp=s myapp.lua
luajit -jp=-s myapp.lua
luajit -jp=vl myapp.lua
luajit -jp=G,profile.txt myapp.lua
```

The following dump features are available. Many of these options can be activated at ones.

Option	Description
f	shows function name (default mode)
F	shows function name with prepend module
l	shows line granularity ('module':'line')
<number>	Stack dump depth (callee<caller - default: 1)
-<number>	inverse stack dump depth (callee>caller
s	split stack dump after first stack level. Implies | *depth* | >= 2
p	show full path for module names
v	VM states (Compiled, Interpreted, C code, Garbage Collector, JIT)
z	show zones (statistics can be grouped in user defined zone)
r	show raw sample counts (Default: percentages)
a	annotate excerpts from source code files
A	annotate complete source code files
G	gives raw output for graphics (time spent per function/VM state)
m<number>	Mminimum sample percentage to be shown (default: 3)
i<number>	sampling interval in milliseconds (default: 10)

Table 6.1: Profiler features

This module can be also used while programming. The `start` function can take 2 arguments, the list of options (describe above) and the output file.

```
local prof = require"jit.p"
prof.start("vf", "file.txt")
  -- Code to analyze here
prof.stop()
```

Statistics can be grouped in user defined *zone*. An exemple is shown below.

```
local zone = require("jit.zone")
zone("MyZone")
  -- Code to analyze here
zone()
```

6.3 Dump mode

This module can be used to debug the JIT compiler itself or to analyse its behaviour when running specific parts of the code. It dumps the code representations and structures used in various compiler stages.

Some examples of using this mode are shown below.

```
luajit -jdump -e "local x=0; for i=1,1e6 do x=x+i end; print(x)"
luajit -jdump=im -e "for i=1,1000 do for j=1,1000 do end end" | less -R
luajit -jdump=is myapp.lua | less -R
luajit -jdump=-bi myapp.lua
luajit -jdump=mbixT,myapp.dump myapp.lua
```

The first argument specifies the dump mode. The second argument gives the output file name (default output is to stdout). The file is overwritten every time the module is started. Different features can be turned on or off with the dump mode. If the mode starts with a '+', the following features are added to the default set of features; a '-' removes them. Otherwise, the features are replaced. The following dump features are available (* marks the default):

Option	Description
`t*`	print a line for each started, ended or aborted trace (see also -jv)
`b*`	dump the traced bytecode
`i*`	dump the IR (intermediate representation)
`r`	augment the IR with register/stack slots
`s`	dump the snapshot map
`m*`	dump the generated machine code
`x`	print each taken trace exit
`X`	print each taken trace exit and the contents of all registers
`a`	print the IR of aborted traces, too
`T`	output format: plain text output
`A`	output format: ANSI-colored text output
`H`	output format: colorized HTML + CSS output

Table 6.2: Dump features

This module can be also used while programming. The `on` function can take 2 arguments, the list of options (describe above) and the output file.

```
local dump = require"jit.dump"
dump.on("tbimT", "outfile.txt")
  -- Code to analyze here
dump.off()
```

This function can have an important role since it gives the opportunity to restrict dump size and locate analysis only on interesting parts of the code.

Bytecode dump

An example of bytecode instruction is shown in the dump below. The official LuaJIT website contains an extensive description of possible bytecode instructions [55].

```
0007  . . CALL     0   0   0  ; comment
```

Column	Description
1st column	bytecode index, numbered by function
2nd column	dots represent the depth (call hierarchy)
3rd-5th columns	bytecode arguments
last column	comment to tie the instruction to the Lua code

Table 6.3: Bytecode intructions

IR dump

Some examples of IR instructions are shown in the dump below. The official LuaJIT website contains an extensive description of possible bytecode instructions [65].

```
....              SNAP   #0   [ ---- ]
0001 rbp      int SLOAD  #2    CI
0002 xmm7  >  num SLOAD  #1    T
....              SNAP   #2   [ ---- 0003 0004 ---- ---- 0004 ]
0003 xmm7  +  num MUL    0002  -1
0010       >  fun EQ     0158  app.lua:298
```

Column	Description
1st column	IR instruction index (numbered per trace)
2nd column	Show where the value is written to, when converted to machine code if the 'r' flags is included (e.g. CPU stack slot, physical CPU stack slot)
3rd column	Instruction flags (">" are locations of guards leading to possible side exits from the trace, "`+`" indicates instruction is left or right PHI operand
4th column	IR type
5th column	IR opcode
6th/7th column	IR operands (e.g. "`%d+`" : reference to IR SSA instruction. "`#`" : prefixes refer to slot numbers, used in SLOADS)

Table 6.4: IR intructions

Snapshot

A snapshot (`SNAP`) stores a consistent view of all updates to the state before an exit. If an exit is taken, the snapshot is used to restore the VM to a consistent state when a trace exits. Each snapshot lists the modified stack slots and the corresponding values. The *n-th* value in the snapshot list represents the index of the IR instruction that wrote in slot number n ("`----`" indicates that the slot has not been modified). Function frames are separated by '|'.

```
....    SNAP   #1   [ ---- ---- ---- ---- 0009 0010 0011 ---- ]
....    SNAP   #1   [ app.mad:120|---- ---- false ]
```

Mcode dump

For the mcode dump each line is composed of two parts, the mcode instruction's address and the corresponding assembler instruction. Some examples of mcode instructions are shown in the dump below.

```
10020ff93  mov dword [0x00041410], 0x1   10020ffca  jg 0x10020001c ->3
10020ff9e  cvttsd2si ebp, [rdx+0x10]     ->LOOP:
10020ffa3  xorps xmm7, xmm7              10020ffd0  xorps xmm6, xmm6
10020ffa6  cvtsi2sd xmm7, ebp            10020ffd3  cvtsi2sd xmm6, ebp
10020ffaa  cmp ebp, +0x5a                10020ffd7  cmp ebp, +0x5a
10020ffad  jz 0x100200014 ->1            10020ffda  jz 0x100200024 ->5
10020ffb3  cmp dword[rdx+0x4],0xffffffff 10020ffe0  addsd xmm7, xmm6
10020ffba  jnb 0x100200018  ->2          10020ffe4  add ebp, +0x01
10020ffc0  addsd xmm7, [rdx]             10020ffe7  cmp ebp, +0x64
10020ffc4  add ebp, +0x01                10020ffea  jle 0x10020ffd0  ->LOOP
10020ffc7  cmp ebp, +0x64                10020ffec  jmp 0x100200028  ->6
```

6.4 Dump mode extension

Apart from the detailed information of bytecode, IR and mcode instructions, the Dump mode of LuaJIT provides information about traces. In fact, it gives an output that is similar to the Verbose mode printing information for each start, end or abortion of a trace.

Our extension of the dump mode introduces more information about traces status (the second line "`TRACE` *`N`* `info ...`"). The output below shows some examples.

```
---- TRACE 1 start app.lua:5
---- TRACE 1 info success trace compilation -- PC=0x7fafd0f79bf0 [61]
---- TRACE 1 stop -> loop

---- TRACE 2 start app.lua:25
---- TRACE 2 info abort penalty pc errno=5 valpenalty=72 -- PC=0x7fd7493e5704 [2]
---- TRACE 2 abort app.lua:4 -- NYI: bytecode 51

---- TRACE 2 start 1/4 app.lua:8
---- TRACE 2 info success trace compilation -- PC=0x7fafd0f79be8
---- TRACE 2 stop -> 1
```

Three lines are printed for each trace:

(i). The first line is emitted when recording starts. It shows the potential trace number, file name and line in the source file from where the trace starts. When the trace recorded is a sidetrace it also prints information on the parent trace and side exit. For instance, in the third example

above 2 is the sidetrace number, 1 is the parent trace from which the sidetrace starts from and 4 is the side-exit number. When the trace is a stitch trace, it display the parent number and the keyword *stitch* (i.e. `TRACE N start N_parent/stitch app.lua:lineNo`)

(ii). The second line (our extension) indicates one of the following situation: (i) trace compilation success; (ii) simple trace abortion; (iii) trace abortion with blacklisting. It also prints the memory address of the code fragment associated to the trace (`PC=0x...`), the index hit in the Hotcount table (in square brackets `[idx]`) and in case of abort also the error number and the penalty value.

(iii). The third line is emitted when a trace creation is concluded. If a trace was successfully generated it shows to what the trace is linked to, e.g. `loop`, `N` (number of another trace), `tail-recursion`, `interpreter`, `stitch`, etc. Otherwise, in case of aborts it shows the file and the source line responsible for the aborts and it prints a message indicating the reason.

The tool was also patched for the diagnosis of the current status of critical data structures used by the JIT for hotpaths detection and blacklisting. In particular, if a special variable is set, it is possible to print the Hotcout table. This is the table in which the counters associated to the hotspots in the code are stored. With this information the user is aware if some collisions occur and what is their impact on trace creation. An example of the output that includes the Hotcout table is shown below.

```
---- HOTCOUNT TABLE
[ 0]=111    [ 1]=111    [ 2]=111    [ 3]=111    [ 4]=111    [ 5]=111    [ 6]=111    [ 7]=111
[ 8]=111    [ 9]=111    [10]=111    [11]=111    [12]=111    [13]=111    [14]=111    [15]=111
[16]=111    [17]=111    [18]=111    [19]=111    [20]=111    [21]=111    [22]=111    [23]=111
[24]=111    [25]=40     [26]=111    [27]=111    [28]=111    [29]=111    [30]=52     [31]=111
[32]=111    [33]=111    [34]=111    [35]=111    [36]=111    [37]=111    [38]=111    [39]=111
[40]=111    [41]=111    [42]=111    [43]=111    [44]=111    [45]=111    [46]=111    [47]=111
[48]=111    [49]=111    [50]=111    [51]=111    [52]=111    [53]=111    [54]=111    [55]=111
[56]=111    [57]=111    [58]=111    [59]=111    [60]=111    [61]=0      [62]=111    [63]=111
---- TRACE 1 start app.lua:5
---- TRACE 1 info success trace compilation -- PC=0x7fafd0f79bf0 [61]
---- TRACE 1 stop -> loop
```

On the other hand, when analysing abort and blacklisting it would be useful to print the Hotpenaly table. This table keeps information on the aborted trace, in order to blacklist fragments when trace creation aborts repeatedly. For each line it prints the memory address of the code fragment from where a trace aborted (`PC=`*`address`*), the penalty value (`val=`*`penalty value`*), the error number (`reason=`*`err number`*). The table has 64 possible entries which are filled through a round-robin index.

```
---- TRACE 2 start app.lua:25
**** HOTPENALTY TABLE penaltyslot=4 (round-robin index)
  [0]:  PC = 7fd7493e56dc  val = 39129   reason =  7
  [1]:  PC = 7fd7493e4dc8  val = 42135   reason =  7
  [2]:  PC = 7fd7493e5198  val = 41161   reason =  7
  [3]:  PC = 7fd7493e5704  val = 18943   reason =  5
  [4]:  PC =             0  val =     0   reason =  0
  [5]:  ...         ...      ...
---- TRACE 2 info abort penalty pc errno=5 valpenalty=37891 -- PC=0x7fd7493e5704 [2]
---- TRACE 2 abort app.lua:4 -- NYI: bytecode 51
```

In the example above the first 3 entries of the table have been blacklisted because their value (`valpenalty=val*2+small_rand_num`) exceeds the maximum penalty value of 60000 (default of LuaJIT). See 4.5.2 for more details on blacklisting in LuaJIT.

6.5 Post-execution Traces Analysis

Along with diagnostic tools, a post-execution analysis on the generated traces can contribute to better understand which parts of the code are more critical for JIT compilation. This analysis can show that traces were or were not organised with an efficient structure. If not, the user should change its source code to make it more JIT-friendly. It is recommended to use programming patterns that can be efficiently compiled by the JIT, and avoid those that would slow down the program execution.

In the context we implemented some scripts in order to collect summary information about traces generated after running an application. The Dump mode can still give useful information, but it prints messages for each potential trace created. In fact, the output of the dump mode can be too large to be analysed (it can be gigabytes of data). Thus, we need to discriminate which part of this huge file should be inspected in more details.

The implemented post-execution trace analysis scripts take as input the output of the Dump mode. Data are filtered and reorganised in a big table containing a row for each hotspot from where the JIT tried to create a trace. The most relevant values stored are: (i) the memory address of the first instruction of the code fragment that was detected as hotspot; (ii) the number of traces that were successfully created from that code fragment (root trace and possible sidetraces attached to it); (iii) the number of aborts or if it was blacklisted; (iv) the list of its parents and the flush number; (v) the file name and the first line of the code fragment in the source code. An example of the output printed is the following:

```
**** TRACE ANALYSIS
PC               Success  Aborts  Blacklist  ParentsList    FileName  Line  Flush
0x7f46176dbb5c   11       2       0          {2,10,11,...}  app1.lua  34    0
0x7f4617660b54   0        11      1          { }            app2.lua  70    0
```

```
0x7f46175b2abc   69      0      0         {4,22,28,...} app3.lua 55   1
```

Once the table is completed we can see which code fragment is more critical. A substantial number of aborts and blacklisting related to a single code fragment can be seen as a red flag, which can deteriorate the overall performance. Also an excessive dimension of the trace tree (many sidetraces attached to a root trace) means that the fragment of code is critical. Once identified the traces that refers to critical code fragments, they should be analysed in details by printing the related bytecode, IR and machine code through the Dump mode. For instance, in the example above the second and third fragments of the table need more investigations. The second fragment aborted many times and it was eventually blacklisted. The third fragment has a high number of success, hence a long chain of sidetraces was attached to the root trace.

Chapter 7

Strategy

The aim of this chapter is to define a strategy to apply in order to produce code in a JIT-friendly style. The relevant questions we should ask ourselves are: *Did I code in such a way that programs execution can profit the most from trace-based just-in-time compilation? How can I ensure that? Do I need any tools to help me in my analysis and decisions to take?*

Specially new users of LuaJIT, which do not really understand how the JIT-compiler works, do not aim to write code in a JIT-friendly style. On the other hand, their first goal is to make a program that works (in the sense that it fulfils its functional requirements), preferably respecting the good practices of software engineering. This is reasonable because firstly we should be able to deliver the program itself, but then we must focus on improving its performance adapting our code in a JIT-friendly style. Even better, we should design from the beginning our software application in a way that programs execution can profit the most from just-in-time compilation.

In the following, we provide a strategy to improve the performances of LuaJIT code. We developed this strategy while analysing a very complex scientific software application which involves loads of computations. We believe that this way to approach coding in LuaJIT can also be used by other people in different contexts. To achieve our goals we basically used the analysis tools provided by Mike Pall with LuaJIT and our Dump extension presented in the previous chapter.

7.1 Profiling

The profiler is a very powerful tool provided by LuaJIT. It is a statistical profiler with very low overhead which allows sampling the currently executing stack and other parameters in regular intervals. Through the profiler, it is

possible to get some insights on programs executions. It shows where the program spends most of the time. In 6.2 we presented how to use the profiler with its options. Here we will show how we used the options of the profiler in order to get useful information.

Through the combination of the options `jp=vl`, the output of the profiler displays the percentage of time spent in each of the VM states (if one of the states does not appears, the time spent in that state is negligible). The profiler also shows for each state the percentage of time spent in each `module:line`.

The example below illustrates a practical case.

Input:

```
./luajit -jp=vl myapp.lua
```

Output:

```
40%  Compiled
  -- 50%  fun1.lua:111
  -- 25%  fun2.lua:222
     ...
25%  Garbage Collector
  -- 33%  fun1.lua:111
  -- 20%  fun3.lua:333
     ...
15%  Interpreted
  -- 35%  fun1.lua:111
  -- 10%  fun2.lua:222
     ...
15%  JIT Compiler
  -- 25%  fun1.lua:111
     ...
 5%  C code
  -- 20%  fun4.lua:444
     ...
```

Here we can see that LuaJIT spent 40% of the time executing machine code and only 15% of the time in the interpreter. This can be view as a good indicator because running compiled code is supposed to be faster than interpreting bytecode instructions. 5% of the time in the C code can be reasonable if the program does not call frequently C functions. 25% of the time spent in the garbage collector might seem a big percentage, but it can be necessary for some situations. 15% of the time spent in the JIT compiler is also a consistent time slice. Ideally, the time spent in the JIT compiler (that is the time spent in recording, generating mcode, etc.) must be kept to the bare minimum, e.g. under 5% of the total execution time.

Moreover, thanks to the option '`l`' we can dig deeper into the specific percentage of time spent in each `module:line` of the VM states. For instance, in the example above the fragment of code at `fun1.lua:111` appears in many

VM states with considerable percentages. This means that it is a critical part of the program which must be analysed in details and eventually optimised.

Thanks to the profiler we can see how the changes in our program may affect the percentage of time spent in each VM state. Ideally, we want to maximise the time spent in the *Compiled* state and minimise the time spent in the *Interpreted*. The time spent in the *C code* is related to the specific application. Finally, we must keep to the bare minimum the time spent in the *JIT compiler* and *Garbage Collector*. It should be noted that this is a general and reasonable guideline to follow, but the overall execution time might be faster even with a slightly disadvantageous configuration (e.g. 70% compiled, 25% Interpreted, 5% JIT Compiler is not necessarily better than 60% compiled, 30% Interpreted, 10% JIT Compiler).

Another interesting combination of options is `jp=Fl`. It can give interesting insights about the specific function and `module:line` where the program spends most of the execution time. In particular, the profiler shows the percentage of time spent in each function and `module:line`.

The example below illustrates a practical case.

Input:

```
./luajit -jp=Fl myapp.lua
```

Output:

```
13%  file1.lua:func1
  -- 67%  file1.lua:866
  -- 16%  file1.lua:886
  --  5%  file1.lua:869
  --  4%  file1.lua:868
10%  file2.lua:func2
  -- 69%  file2.lua:194
  -- 13%  file2.lua:192
  -- 10%  file2.lua:195
  --  7%  file2.lua:191
9%  file3.lua:func3
  -- 68%  file3.lua:714
  -- 11%  file3.lua:736
  --  5%  file3.lua:718
  --  3%  file3.lua:733
  --  3%  file3.lua:734
  --  3%  file3.lua:735
...
```

To conclude our discussion about the profiler, we want to highlight the fact that when running the profiler LuaJIT calls a function to flush all traces (see `lj_profile.c` for more details).

7.2 Dump mode and post-execution analysis

The profiler is a very powerful tool that can help us to detect critical parts of the code where programs spend most of the time. It also shows how the changes we make to the code impact on the time spent in each state of the VM. However, what is missing in the profiler is the information about the traces generated by the JIT compiler. Having information about traces can guide us to change our code in a "good" way so that the compiler will generate traces more efficiently.

The Dump mode provides useful information about traces at different levels of granularity. In this context, we used the Dump tool with our small patch as explained in Sec. 6.4.

The example below illustrates a practical case.

Input:

```
./LuaJITpatched/src/luajit -jdump=t myapp.lua -o nil >output.txt 2<&1
```

Output:

```
---- TRACE 1 start file1.lua:328
---- TRACE 1 info abort penalty pc errno=7 valpenalty=144 -- PC=0x7f10791ab78c [36]
---- TRACE 1 abort file1.lua:335 -- NYI: bytecode 50

---- TRACE 1 start madl_utest.lua:1921
---- TRACE 1 info success trace compilation -- PC=0x7f1079126a58 [23]
---- TRACE 1 stop -> loop

---- TRACE 2 start file1.lua:328
---- TRACE 2 info abort penalty pc errno=7 valpenalty=298 -- PC=0x7f10791ab78c [36]
---- TRACE 2 abort file1.lua:335 -- NYI: bytecode 50

---- TRACE 2 start file1.lua:134
---- TRACE 2 info success trace compilation -- PC=0x7f10791dcf14 [6]
---- TRACE 2 stop -> loop

---- TRACE 3 start 2/1 file1.lua:135
---- TRACE 3 info success trace compilation -- PC=0x7f10791dcef0
---- TRACE 3 stop -> 2

---- TRACE flush

---- TRACE 1 start file2.lua:182
---- TRACE 1 info success trace compilation -- PC=0x7f1079193c48 [19]
---- TRACE 1 stop -> return

---- TRACE 2 start file2.lua:2244
---- TRACE 2 info success trace compilation -- PC=0x7f107908c3b0 [45]
```

The goal is to print in the file`output.txt` information of the traces created by the JIT compiler. The Dump mode is set to `jdump=t` which prints a line for each started, ended or aborted trace; the option `-o nil` avoids any print from the application in the `output.txt` file; `2<&1` redirects all the output on the same standard.

The output of the dump mode can be too large to be analysed (it can be gigabytes of data). Thus, we need to discriminate which part of this huge file should be inspected in more details.

Once the `output.txt` file is created we used a post-execution script `TraceAnalyzer.lua` in order to collect summary information about traces as described in Sec. 6.5. Data are filtered and reorganised from the file `output.txt` in a big table containing a row for each hotspot from where the JIT tried to create a trace. Here we can decide the columns we want to print. In the example below the output shows: (i) the memory address of the first instruction of the code fragment that was detected as hotspot; (ii) the nature of the trace, i.e. root, side or stitch; (iii) the link at the end of the trace, e.g. the trace number of its parent, `nil` means no connection because the trace is aborted, etc.; (iv) the number of aborts or if it was blacklisted; (v) the file name and the first line of the code fragment in the source code; (v) the error file name, the first line of the code fragment in its source code and the error number; (vi) the number of flushes that occurred before the creation of the trace.

Input:

```
./luajit TracesAnalyser.lua >Result.txt
```

Output:

```
**** TRACE ANALYSIS
PC              What  Link Aborts Blacklist FileName Line Errfile  Errline Errno Flush
0x7f46176dbb5c  side  4    2      0         app1.lua 34   app9.lua 23      10    0
0x7f4617660b54  root  nil  11     1         app2.lua 70   app5.lua 90      8     0
0x7f46175b2abc  side  12   0      0         app3.lua 55   app5.lua 57      10    1
```

Once the table is completed we can see which code fragment is more critical. A substantial number of aborts and blacklisting related to a single code fragment can be seen as a red flag, which can deteriorate the overall performance. This can be seen as a general indicator but, specially in large and complex applications, aborts and/or blacklisting will always occur. It should be noted that the fact of identifying blacklisted fragments of code is possible only thanks to our patch of the Dump mode since the standard version gives only information about aborts.

Once identified the traces that refers to critical code fragments, they should be analysed in details by printing the related bytecode, IR and machine code through the Dump mode (options `bim`). In order to do that we must activate the dump mode only in the fragments of code that we classified as "critical".

The example below illustrates a practical case.

```
local dump = require "jit.dump"
```

```
dump.on ( "bim", "outfile.txt")
  -- code to analyse here
dump.off( )
```

A explained in the previous chapter post-execution analysis on the generated traces can contribute to better understand which parts of the code are more critical for JIT compilation. This analysis can show that traces were or were not organised with an efficient structure. If not, the user should change its source code to make it more JIT-friendly.

7.3 Tuning LuaJIT parameters

Another key aspect which can improve the performances of LuaJIT consists of tuning the parameter of the JIT compiler. LuaJIT was originally designed by Mike Pall to be run in embedded applications with memory limitations, hence the default values of the JIT compiler parameters respect these memory constraints.

The table below shows the JIT compiler parameters with description and default values.

Parameter	**Default**	**Description**
maxtrace	1000	Max. number of traces in the cache
maxrecord	4000	Max. number of recorded IR instructions
maxirconst	500	Max. number of IR constants of a trace
maxside	100	Max. number of side traces of a root trace
maxsnap	500	Max. number of snapshots for a trace
hotloop	56	Number of iterations to detect a hot loop or hot call
hotexit	10	Number of taken exits to start a side trace
tryside	4	Number of attempts to compile a side trace
instunroll	4	Max. unroll factor for instable loops
loopunroll	15	Max. unroll factor for loop ops in side traces
callunroll	3	Max. unroll factor for pseudo-recursive calls
recunroll	2	Min. unroll factor for true recursion
sizemcode	32	Size of each machine code area in KBytes (Windows:64K)
maxmcode	512	Max. total size of all machine code areas in KBytes

Table 7.1: JIT compiler parameters

In chapter 2 we mentioned OpenResty [49], a relevant software that integrates LuaJIT. In this application some of the default parameters were changed: `maxtrace=8000`, `maxrecord=16000, minstitch=3, maxcode=40960`.

Also in our complex scientific application that involves loads of computations we observed a faster and more stable behaviour when changing some of the default parameters. In particular our final configuration includes: `maxtrace=8000`, `maxrecord=16000, loopunroll=7, maxcode=65536`.

It is clear that in both these applications extending the limitations in terms of memory constraints leads to an improvement in the performance.

In order to change the default parameters, you must run LuaJIT with the option `-Oparam=value`. Once you find the most appropriate configuration of parameters for your specific application you can then recompile LuaJIT with the values of the new parameters.

It should be noted that this is a very delicate task. The default values were defined based on heuristics by Mike Pall, who actually commented this subject in a post on the LuaJIT Github project saying: *"The trace heuristics are a very, very delicate matter. They interact with each other in many ways and it will only show in very specific code. This is hard to tune unless one is willing to do lots of benchmarking and stare at generated code for hours. That's how the current heuristics have been derived. A major project for someone with enough time at their hands"*[1].

Here we want to describe how we tuned the `loopunroll` parameter for our specific application so that we provide a concrete example for the reader. Thanks to the script `TraceAnalyser.lua`, which processes the output of the patched version of the Dump mode, we realised that an enormous number of trace aborts were caused by `loop unroll limit reached` (`errno=10`). The abort `loop unroll limit reached` means that the JIT compiler has applied the loop unrolling optimisations too many times in a given trace. A first solution would be to restructure the code into a few tight loops instead of one big loop (or vice-versa) or another option is to change to loopunroll limit.

We realised that `15` (the default value of `loopunroll`) was not appropriate for our specific application. Incrementing this value (e.g. `loopunroll=30`) made the performance of the overall application worst. Thus, we decremented the default value trying different alternatives. Finally, we decided to go for `loopunroll=7` as our optimal value because we observed a faster and more stable behaviour of the overall application.

Citing again Mike Pall from a post on the LuaJIT Github project: *The loop unrolling heuristics are not ideal for all situations. Some dependent projects change the loop unrolling limits*[1].

[1]Conversation on the LuaJIT Github project: `https://github.com/LuaJIT/LuaJIT/issues/11`

Appendix A

Values in dumps

In this appendix, we presents possible values that can be shown in dumps and their corresponding meaning.

Table A.1: Possible values for `[link]` (see *jit_trlinkname* and *TraceLink* in `lib_jit.[c,h]`)

value	Meaning
none	Incomplete trace. No link, yet.
root	Link to other root trace.
loop	Loop to same trace.
tail-recursion	Tail-recursion.
up-recursion	Up-recursion.
down-recursion	Down-recursion.
interpreter	Fallback to interpreter (stop a side trace record due to maximum reached see *sidecheck:* in lj_record.c)
return	Return to interpreter.
stitch	trace stitching.

Table A.2: Possible value for SLOAD argument (see `lj_ir.h`)

P	IRSLOAD_PARENT	Coalesce with parent trace.
F	IRSLOAD_FRAME	Load 32 bits of ftsz.
T	IRSLOAD_TYPECHECK	Needs type check.
C	IRSLOAD_CONVERT	Number to integer conversion.
R	IRSLOAD_READONLY	Read-only, omit slot store.
I	IRSLOAD_INHERIT	Inherited by exits/side traces.

Table A.3: Possible value for XLOAD argument (see `lj_ir.h`)

R	IRXLOAD_READONLY	Load from read-only data.
V	IRXLOAD_VOLATILE	Load from volatile data.
U	IRXLOAD_UNALIGNED	Unaligned load.

Table A.4: Possible value for FLOAD or FREF argument (see `lj_ir.h`)

str.len	IRFL_STR_LEN	String length
func.env	IRFL_FUNC_ENV	function's environment for up-values
func.pc	IRFL_FUNC_PC	PC of the function's prototype
func.ffid	IRFL_FUNC_FFID	Function id
thread.env	IRFL_THREAD_ENV	Thread environment for up-values
tab.meta	IRFL_TAB_META	Metatable
tab.array	IRFL_TAB_ARRAY	Table array part
tab.node	IRFL_TAB_NODE	Table hash part
tab.asize	IRFL_TAB_ASIZE	Size of array part
tab.hmask	IRFL_TAB_HMASK	Size of hash part - 1
tab.nomm	IRFL_TAB_NOMM	Negative cache for fast metamethods bitmap, marking absent fields of the metatable
udata.meta	IRFL_UDATA_META	udata metatable
udata.udtype	IRFL_UDATA_UDTYPE	see UDTYPE table
udata.file	IRFL_UDATA_FILE	udata payload
cdata.ctypeid	IRFL_CDATA_CTYPEID	cdata's ctypeid
cdata.ptr	IRFL_CDATA_PTR	cdata payload
cdata.int	IRFL_CDATA_INT	cdata payload
cdata.int64	IRFL_CDATA_INT64	cdata payload
cdata.int64_4	IRFL_CDATA_INT64_4	cdata payload

Table A.5: Possible value for userdata types (see `lj_obj.h`)

UDTYPE_USERDATA	Regular userdata.
UDTYPE_IO_FILE	I/O library FILE.
UDTYPE_FFI_CLIB	FFI C library namespace.

Table A.6: Possible value for FPMATH argument (see `lj_ir.h`)

floor	FPM_FLOOR
ceil	FPM_CEIL
trunc	FPM_TRUNC
sqrt	FPM_SQRT
exp	FPM_EXP
exp2	FPM_EXP2
log	FPM_LOG
log2	FPM_LOG2
log10	FPM_LOG10
sin	FPM_SIN
cos	FPM_COS
tan	FPM_TAN

Table A.7: Possible value for BUFHDR argument (see `lj_ir.h`)

IRBUFHDR_RESET	Reset buffer
IRBUFHDR_APPEND	Append to buffer

Table A.8: Possible value for TOSTR argument (see `lj_ir.h`)

INT	IRTOSTR_INT	Convert integer to string.
NUM	IRTOSTR_NUM	Convert number to string.
CHAR	IRTOSTR_CHAR	Convert char value to string.

Appendix B

DynASM: Assembler

DynASM [5] is a Dynamic Assembler for code generation engines. It has been developed primarily as a tool for LuaJIT and its source code can be found in the *dynasm* folder of the LuaJIT project [70]. It is currently used in LuaJIT 2 has a tool to write the fast VM interpreting the bytecode. It supports the following platforms: x86, x64, ARM, PowerPC, and MIPS. The official documentation for DynASM [5] is extremely spartan, but an unofficial documentation with tutorials by Peter Cawley is available at [46]. In this appendix we present the minimum amount of information to be able to read and understand a DynASM file (`*.dasc`).

Main directives

Directive	Description
`.arch`	specifies the architecture of the assembly code
`.type name, ctype [, default_reg]`	makes easier to manipulate registers of type *ctype**. The provided syntactic sugar is depicted in Table B.2
`.macro [...] .endmacro`	create a multi-lines macro instruction that can be invoked as a normal instruction and where arguments will be substituted
`.define`	defines a prepreprocessor substitution
`.if [...] .elif [...] .else [...] .endif`	Preprocessor conditional construct similar as C preprocessor

Table B.1: Dynasm main directives

Sugar	Expansion
`#name`	sizeof(ctype)
`name:reg->field`	[reg + offsetof(ctype,field)]
`name:reg[imm32]`	[reg + sizeof(ctype)*imm32]
`name:reg[imm32].field`	[reg + sizeof(ctype)*imm32 + offsetof(ctype,field)]
`name:reg...`	[reg + (int)(ptrdiff_t)&(((ctype*)0)...)]

Table B.2: Syntactic sugar provided by `.type` macro

Line markers

Typical DynASM lines that emit assembler instructions must start with a vertical bare ("|"). If you want to emit some lines of C code, but you still want DynASM's preprocessor to do substitutions, they must start with a double vertical bar ("||"). Finally, lines with no starting markers are completely untouched and discarded by DynASM. It should be noted that lines of C code that has to be inlined with a macro must start with a double vertical bar.

Labels

There are different kind of labels. The first category is the "global labels" that have two categories, "static label" (|->`name:`) and "dynamic labels" (|=>`imm32:`). These labels are uniques in a DynASM file. The second category is the "local labels" that use a single digit from 1 to 9 (|`i:`). They can be defined multiple times in the same DynASM file. They are used by jump instructions with of the syntax <`i` or >`i`. They respectively point to the most recent, the next definition of `i` as the jump target.

Bibliography

[1] Dario d'Andrea. "Behavioural Analysis of Tracing JIT Compiler Embedded in the Methodical Accelerator Design Software". In: (2019).

[2] PUC-Rio Tecgraf. *The Programming Language Lua*. URL: `http://www.lua.org/`.

[3] Mike Pall. *LuaJIT 2*. 2012. URL: `http://luajit.org`.

[4] Aurelien Bloch. "First Performance Analysis of MAD-NG Embedded Tracing JIT Compiler". In: (2018).

[5] Mike Pall. *DynASM*. 2012. URL: `https://luajit.org/dynasm.html`.

[6] Matthew Arnold et al. "A survey of adaptive optimization in virtual machines". In: *Proceedings of the IEEE* 93.2 (2005), pp. 449–466.

[7] Antonio Cuni. "High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages". PhD thesis. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione ..., 2010.

[8] Wikipedia contributors. *Pareto principle — Wikipedia, The Free Encyclopedia*. 2019.

[9] Philipp Hoschka and Christian Huitema. "Control flow graph analysis for automatic fast path implementation". In: *Second IEEE workshop on the architecture and Implementation of high performance communication subsystems*. 1993.

[10] Thomas Schilling. "Trace-based Just-in-time Compilation for Lazy Functional Programming Languages". PhD thesis. University of Kent, 2013.

[11] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005. Chap. 2.

[12] Richard E Hank, Wen-Mei W Hwu, and B Ramakrishna Rau. "Region-based compilation: An introduction and motivation". In: *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE. 1995, pp. 158–168.

[13] Derek Bruening and Evelyn Duesterwald. "Exploring optimal compilation unit shapes for an embedded just-in-time compiler". In: *In Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*. Citeseer. 2000.

[14] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. *Transparent dynamic optimization: The design and implementation of Dynamo*. Tech. rep. HP Laboratories Cambridge, 1999.

[15] Evelyn Duesterwald and Vasanth Bala. "Software profiling for hot path prediction: Less is more". In: *ACM SIGOPS Operating Systems Review*. Vol. 34. 5. ACM. 2000, pp. 202–211.

[16] Andreas Gal, Christian W Probst, and Michael Franz. "HotpathVM: an effective JIT compiler for resource-constrained devices". In: *Proceedings of the 2nd international conference on Virtual execution environments*. ACM. 2006, pp. 144–153.

[17] Carl Friedrich Bolz et al. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM. 2009, pp. 18–25.

[18] David Hiniker, Kim Hazelwood, and Michael D Smith. "Improving region selection in dynamic optimization systems". In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2005, pp. 141–154.

[19] Andreas Gal et al. "Trace-based just-in-time type specialization for dynamic languages". In: *ACM Sigplan Notices* 44.6 (2009), pp. 465–478.

[20] Luke Gorrie. *RaptorJIT*. 2017. URL: `https://github.com/raptorjit/raptorjit`.

[21] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[22] Andreas Gal and Michael Franz. *Incremental dynamic code generation with trace trees*. Tech. rep. Citeseer, 2006.

[23] James G Mitchell. "The design and construction of flexible and efficient interactive programming systems." PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, 1970.

[24] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: a transparent dynamic optimization system”. In: (2000).

[25] John Aycock. “A brief history of just-in-time”. In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[26] Dean Deaver, Rick Gorton, and Norm Rubin. “Wiggins/Redstone: An on-line program specializer”. In: *Proceedings of the IEEE Hot Chips XI Conference*. 1999.

[27] Michael Gschwind et al. “Dynamic and transparent binary translation”. In: *Computer* 33.3 (2000), pp. 54–59.

[28] Cindy Zheng and Carol Thompson. “PA-RISC to IA-64: Transparent execution, no recompilation”. In: *Computer* 33.3 (2000), pp. 47–52.

[29] Gregory T Sullivan et al. “Dynamic native optimization of interpreters”. In: *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. ACM. 2003, pp. 50–57.

[30] Mozilla Foundation. *SpiderMonkey*. URL: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[31] Mason Chang et al. “Tracing for web 3.0: trace compilation for the next generation web applications”. In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM. 2009, pp. 71–80.

[32] Adobe System Inc. *ActionScript 3.0 overview*. URL: `https://www.adobe.com/devnet/actionscript/articles/actionscript3%5C_overview.html`.

[33] Ecma International. *Standard ECMA-262*. 1999. URL: `https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[34] Andreas Gal. “Efficient Bytecode Verification and Compilation in a Virtual Machine Dissertation”. PhD thesis. PhD thesis, University Of California, Irvine, 2006.

[35] Andreas Gal et al. *Making the compilation “pipeline” explicit: Dynamic compilation using trace tree serialization*. Tech. rep. Technical Report 07-12, University of California, Irvine, 2007.

[36] Mason Chang et al. *Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference*. Tech. rep. Citeseer, 2007.

[37] Mason Chang et al. “The impact of optional type information on jit compilation of dynamically typed languages”. In: *ACM SIGPLAN Notices* 47.2 (2012), pp. 13–24.

[38] Carl Friedrich Bolz et al. “Allocation removal by partial evaluation in a tracing JIT”. In: *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM. 2010, pp. 43–52.

[39] Carl Friedrich Bolz and Laurence Tratt. “The impact of meta-tracing on VM design and implementation”. In: *Science of Computer Programming* (2013).

[40] Carl Friedrich Bolz et al. “Meta-tracing makes a fast Racket”. In: *Workshop on Dynamic Languages and Applications*. 2014.

[41] Carl Friedrich Bolz. “Meta-tracing just-in-time compilation for RPython”. PhD thesis. Heinrich Heine University Düsseldorf, 2012.

[42] Håkan Ardö, Carl Friedrich Bolz, and Maciej FijaBkowski. “Loop-aware optimizations in PyPy’s tracing JIT”. In: *ACM SIGPLAN Notices*. Vol. 48. 2. ACM. 2012, pp. 63–72.

[43] Maarten Vandercammen. “The Essence of Meta-Tracing JIT Compilers”. In: (2015).

[44] Spenser Bauman et al. “Pycket: a tracing JIT for a functional language”. In: *ACM SIGPLAN Notices*. Vol. 50. 9. ACM. 2015, pp. 22–34.

[45] Michael Bebenita et al. “SPUR: a trace-based JIT compiler for CIL”. In: *ACM Sigplan Notices* 45.10 (2010), pp. 708–725.

[46] Peter Cawley. *The unofficial DynASM documentation*. URL: `https://corsix.github.io/dynasm-doc/`.

[47] Mike Pall. *LuaJIT 2.0 intellectual property disclosure and research opportunities*. 2009. URL: `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[48] Luke Gorrie. *Studio*. 2017. URL: `https://github.com/studio/studio/`.

[49] OpenResty Inc. *OpenResty*. URL: `https://openresty.org/`.

[50] Mike Pall. *LuaJIT New Garbage Collector wiki page*. 2017. URL: `http://wiki.luajit.org/New-Garbage-Collector`.

[51] Doug Lea. *Doug Lea Memory Allocator article*. 2018. URL: `http://g.oswego.edu/dl/html/malloc.html`.

[52] Doug Lea. *Doug Lea Memory Allocator implementation*. 2018. URL: ftp://g.oswsego.edu/pub/misc/malloc.c.

[53] Mike Pall. *LuaJIT Extensions*. 2018. URL: http://luajit.org/extensions.html.

[54] Mike Pall. *LuaJIT Bit Operations Module*. 2018. URL: http://bitop.luajit.org/.

[55] Mike Pall. *LuaJIT 2.0 Bytecode Instructions*. URL: http://wiki.luajit.org/Bytecode-2.0.

[56] Mike Pall. *FFI motivation and use*. 2018. URL: http://luajit.org/ext%5C_ffi.html.

[57] Mike Pall. *FFI tutorial*. 2018. URL: http://luajit.org/ext%5C_ffi%5Ctutorial.html.

[58] Mike Pall. *FFI API documentation*. 2018. URL: http://luajit.org/ext%5C_ffi%5C_api.html.

[59] Mike Pall. *FFI Semantics*. 2018. URL: http://luajit.org/ext%5C_ffi%5C_semantics.html.

[60] Peter Cawley. *Reflection library for ctypes*. 2018. URL: https://github.com/corsix/ffi-reflect.

[61] Peter Cawley. *FFI reflect documentation*. 2018. URL: http://corsix.github.io/ffi-reflect.

[62] Mike Pall. *LuaJIT Wiki*. URL: http://wiki.luajit.org/Home.

[63] Mike Pall. *LuaJIT optimization wiki page*. 2018. URL: http://wiki.luajit.org/Optimizations.

[64] Mike Pall. *LuaJIT Allocation Sinking Optimization wiki page*. 2017. URL: http://wiki.luajit.org/Allocation-Sinking-Optimization%5C#implementation%5C_assembler-backend%5C_snapshot-allocations.

[65] Mike Pall. *LuaJIT 2.0 SSA IR*. URL: http://wiki.luajit.org/SSA-IR-2.0.

[66] Mike Pall. *Mail: LuaJIT 2.0 intellectual property disclosure and research opportunities*. 2017. URL: http://lua-users.org/lists/lua-l/2009-11/msg00089.html.

[67] Roberto Ierusalimschy. "Programming in Lua, Fourt Edition". In: Lua.org, 2016. Chap. 22.

[68] *LUA Reference Manual*. URL: https://www.lua.org/manual/.

[69] *Static Single Assignment Book*. URL: http://ssabook.gforge.inria.fr/latest/book.pdf.

[70] Mike Pall. *LuaJIT source code*. 2017. URL: https://github.com/LuaJIT/LuaJIT.