

UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale in Calcolatori Elettronici II

Behavioural Analysis of Tracing JIT Compiler Embedded in the Methodical Accelerator Design Software

Anno Accademico 2018/2019

relatori

Ch.mo prof. Nicola Mazzocca

Ch.mo prof. Pasquale Arpaia

correlatore

Ing. Laurent Deniau PhD

candidato

Dario d'Andrea

matr. M63000695



Acknowledgements

Firstly, I would like to thank my supervisor at CERN, Laurent Deniau, for his daily support and his useful suggestions throughout the work described in this thesis. I would like to express my gratitude to both my university supervisors, Nicola Mazzocca and Pasquale Arpaia, for their helpfulness during this work and for their support during the past years at university. I feel privileged of being allowed to work with such inspiring mentors.

This thesis would not have been possible without the help from the community of the LuaJIT project including all the useful insights contained in its mailing list, specially by its author, Mike Pall, who worked for many years accomplishing an amazing job.

A special acknowledgement should be addressed to my family. I thank my father Guido and my mother Leda who guided me with love during my education and my life. I am grateful to my brother Fabio, my grandmother Tina, and my uncle Nicola, for their support during the past years. I also want to remember my uncle Bruno who inspired me for my academic career.

I wish to express my deepest gratitude to Alicia for her unconditional encouragement. Her support was essential to overcome difficulties and to encourage me in what I pursued. I am grateful to Diego, with whom I shared my entire university course, not only for the countless days of study, but also for being a true friend.

Last but not least, I want to thank all my friends from Napoli, the guys from BEST, people from the Erasmus in the Netherlands and, most recently, my friends from CERN in Geneva with who I shared the happiest and most difficult moments during the writing of this thesis.

Ringraziamenti

In primo luogo vorrei ringraziare il mio supervisore del CERN, Laurent Deniau, per il suo supporto quotidiano e i consigli estremamente utili durante il lavoro descritto in questa tesi. Vorrei inoltre ringraziare entrambi i miei relatori universitari, Nicola Mazzocca e Pasquale Arpaia, per la loro disponibilità durante il lavoro di tesi e per il loro supporto negli anni trascorsi all'università. Mi sento onorato di aver avuto tali mentori da cui trarre ispirazione.

Questa tesi non sarebbe stata possibile senza il sostegno della comunità del progetto di LuaJIT con le utili opinioni scambiate nella sua mailing list, specialmente dal suo autore, Mike Pall, che ha dedicato numerosi anni nel realizzare un lavoro formidabile.

Un ringraziamento speciale va alla mia famiglia. Ringrazio mio padre Guido e mia madre Leda che mi hanno guidato con amore nella mia educazione e nella mia vita. Sono grato a mio fratello Fabio, a mia nonna Tina e a mio zio Nicola per il loro straordinario sostegno negli anni trascorsi. Vorrei anche ricordare mio zio Bruno da cui ho preso ispirazione per la mia carriera accademica.

Desidero esprimere la mia più profonda gratitudine ad Alicia per il suo incoraggiamento incondizionato. Il suo supporto è stato fondamentale nel superare le difficoltà e nel raggiungere i miei obiettivi. Vorrei ringraziare Diego con cui ho condiviso il mio intero percorso universitario, non solo per gli innumerevoli giorni di studio, ma anche per essere un vero amico.

Ultimo ma non meno importante, voglio ringraziare tutti gli amici da Napoli, i ragazzi di BEST, gli amici dell'Erasmus in Olanda e, più di recente, i colleghi del CERN a Ginevra con cui ho condiviso i momenti più felici e difficili nella scrittura di questa tesi.

Abstract

This thesis investigates with a structured and precise analysis the behaviour of programs execution during trace-based just-in-time (JIT) compilation. The work was carried out in the context of the tracing just-in-time compiler LuaJIT embedded in the Next Generation of the Methodical Accelerator Design software (MAD-NG).

MAD software series provides a scripting language, which is a *de facto standard* to describe particle accelerators, simulate beam dynamics, and optimise beam optics at CERN. It has to process a massive quantity of data and operations with user-defined expressions during simulation, hence it needs to be very efficient. A technology that fulfils these demands is LuaJIT, a trace-based just-in-time compiler for the Lua programming language. Trace-based just in time compilation is a technique used for dynamic interpreted languages. It consists of recording a linear path of frequently executed operations, so-called *trace*, compiling it to native machine code and executing it.

To accomplish this work we proposed some extensions for the existing analysis tools provided by LuaJIT and we realised some post-processing analysis to inspect the traces generated by the JIT during the execution of a program. It is necessary to understand all the mechanisms performed by a tracing JIT compiler under the hood in order to better profit from trace-based just-in-time compilation. The consciousness of the internal mechanisms behind a tracing JIT compiler leads to take high-level software decision in JIT-friendly style.

Contents

1	Introduction	1
1.1	CERN	1
1.2	CERN accelerator complex	2
1.3	Methodical Accelerator Design software	3
1.3.1	MAD-X	3
1.3.2	MAD-NG	4
1.4	Software design decision: LuaJIT	4
1.5	Thesis statement and overview	5
2	Background	7
2.1	Just-in-time compilation	7
2.2	Compilation units	8
2.3	Trace-based Just-in-time Compilation	10
2.3.1	Identifying trace headers	14
2.3.2	Hotpath detection	15
2.3.3	Trace recording	17
2.3.4	Abort and blacklisting	18
2.3.5	Compiling traces	20
2.3.6	Trace exit	21
2.3.7	Sidetraces	22
3	Related works	26
3.1	Early Tracing JITs	26
3.2	Recents Tracing JITs	27

3.3	An introduction to LuaJIT	30
3.3.1	Lua	30
3.3.2	LuaJIT overview	30
4	Building a trace in LuaJIT	33
4.1	Hotpaths detection	34
4.1.1	Architecture specific implementation	37
4.1.2	Hotcount collisions	38
4.1.3	Memory address randomisation	39
4.2	Recording	39
4.2.1	Trace compiler state machine	40
4.2.2	Start recording	42
4.2.3	Recording	43
4.2.4	Ending recording	44
4.3	Assemble trace	45
4.4	Abort	47
4.5	Blacklisting	51
4.5.1	Blacklisting cancer	56
5	LuaJIT traces investigation	59
5.1	Introduction	59
5.2	Essential cases	59
5.2.1	Empty loop	60
5.2.2	Loop with assignment	62
5.2.3	Loop with if-statements	65
5.2.4	Nested loop	68
5.3	Loop with two if-statements	71
5.3.1	Case 1	71
5.3.2	Case 2	75
5.3.3	Case 3	79
5.3.4	Case 4	82

5.4	Nested loop with more inner loops	86
5.5	Function	90
5.5.1	Non-tail recursive function	90
5.5.2	Tail recursive function	92
6	Conclusions	94
6.1	Overview	94
6.2	Difficulties and strategy	95
6.3	Results	96
6.4	Future works	97
A	Analysis Tools	98
A.1	Verbose mode	98
A.2	Profiler	99
A.3	Dump mode	100
A.4	Dump mode extensions	104
A.5	Post-execution Traces Analysis	106

List of Figures

1.1	CERN accelerators complex (source [3])	2
2.1	Stages of execution for a VM with tracing JIT	11
2.2	Hotpath detection	16
2.3	Example of loop	18
2.4	Blacklisting	20
2.5	Sidetrace creation	23
2.6	Loop	24
2.7	Traces generated	24
3.1	Schematic view of LuaJIT internals (source [14])	31
4.1	Hotcount decrement for loops	36
4.2	Trace compiler state machine	41
4.3	Start recording	43
4.4	Recording a bytecode instruction	44
4.5	Assemble trace	45
4.6	Stop tracing	46
4.7	Throw error	50
4.8	Abort	50
4.9	Penalization and blacklisting	55
5.1	Trace flow diagram empty loop	61

5.2	Trace flow diagram loop with assignment	64
5.3	Trace flow diagram loop with if-statement	67
5.4	Trace flow diagram nested loop	70
5.5	Trace flow diagram loop with 2 if-statements Example 1	74
5.6	Trace flow diagram loop with 2 if-statements Example 2	78
5.7	Trace flow diagram loop with 2 if-statements Example 3	81
5.8	Trace flow diagram loop with 2 if-statements Example 4	85
5.9	Trace flow diagram loop with 2 inner loops	88
5.10	Trace flow diagram loop with 3 inner loops	90

List of Tables

4.1	Bytecode instructions for hotpaths detection	34
4.2	Example snapshot of an Hotcount Table	35
4.3	Trace compiler state encoding	40
4.4	Bytecode instructions to force JIT-compiled trace execution	46
4.5	Trace compiler error messages	48
4.6	Parameters of the JIT compiler	51
4.7	Bytecode instructions to force interpretation	52
4.8	Exemplifying snapshot of the Hot Penalty Table	53
A.1	Profiler features	100
A.2	Dump features	101
A.3	Bytecode intructions	102
A.4	IR intructions	103

Chapter 1

Introduction

1.1 CERN

The European Organisation for Nuclear Research, known as CERN (Conseil européen pour la recherche nucléaire) is one of the world's largest and most famous centres for scientific research. Originally established in 1954 by 12 founding states it currently counts 23 member states. Thanks to the cooperation between nations, universities and scientists, CERN tries to push the limits of knowledge. Its primary mission is related to fundamental physics in which it aims to discover what the Universe is made of and how it works. One of its most recent and famous achievement was the discovery of the Higgs boson in 2012 [1, 2].

Since particle physics demands the ultimate in performance, CERN laboratory also plays a key role in developing cutting-edge technologies in various fields of application from materials science to computing. The most-known CERN technology in computing is the World Wide Web, developed by Tim Berners-Lee in 1989 while employed at CERN.

This thesis was carried out in the context of a project at CERN for the Methodical Accelerator Design software (MAD) supported by the Accelerators and Beam Physics group in the Beams department.

1.2 CERN accelerator complex

At present, CERN hosts the largest particle accelerator laboratory in the world which is located at the Franco-Swiss border near Geneva. The CERN accelerators complex (or chain) is shown in Fig. 1.1

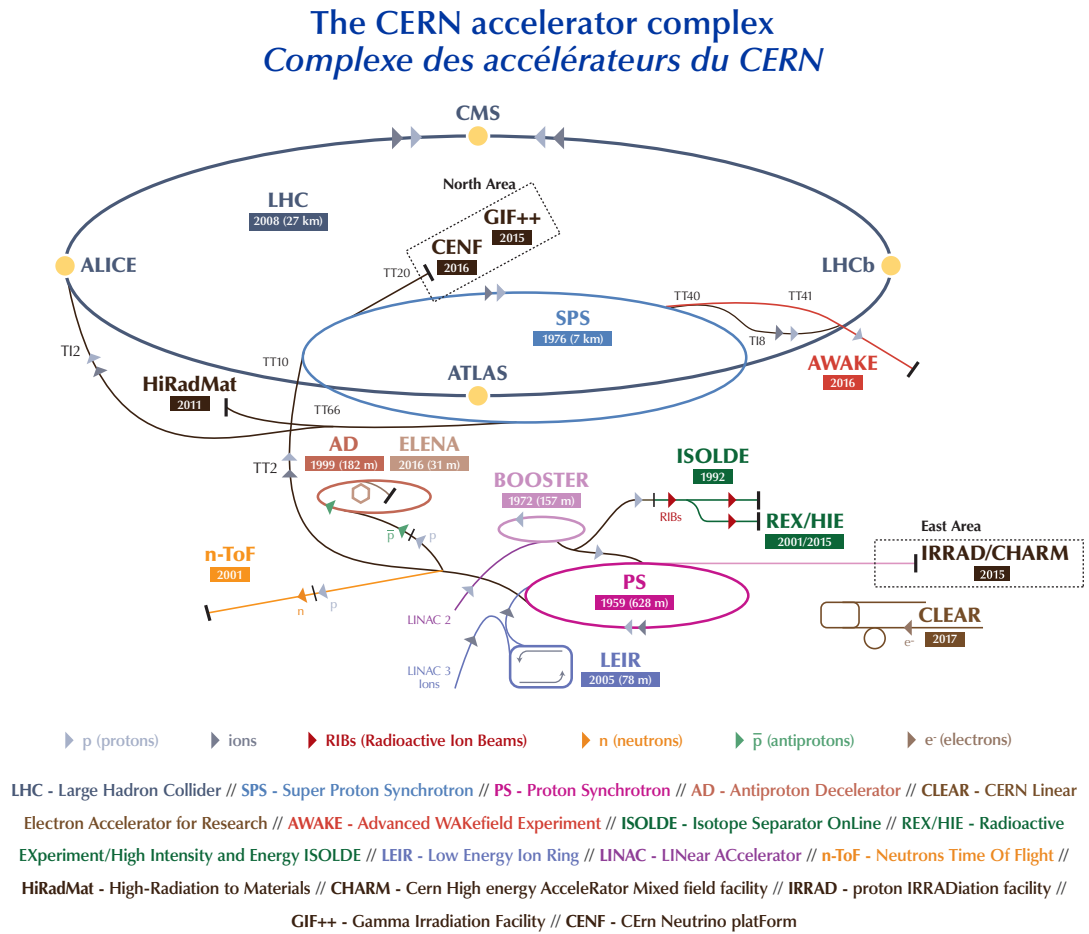


Figure 1.1: CERN accelerators complex (source [3])

CERN's accelerator complex consists of a series of particle accelerators that can reach increasingly higher energies. Each accelerator increases the speed of a particle beam, before injecting it into the next one in the succession.

The overall process starts by obtaining protons from hydrogen atoms removing

electrons. Protons are injected from a linear accelerator (LINAC) into the Booster, followed by the Proton Synchrotron (PS), then the Super Proton Synchrotron (SPS) and finally the Large Hadron Collider (LHC). The LHC is the world's largest and the most powerful particle collider with a ring of 27 km long. In this accelerator, beams circulate in opposite directions and they are forced to collide at four points around the accelerator ring where the four particle detectors (ATLAS, CMS, ALICE and LHCb) are located.

1.3 Methodical Accelerator Design software

The Methodical Accelerator Design software (MAD) [4] is a project with a long history that aims to be at the forefront of computational physics in the field of particle accelerator design and simulation. MAD provides a scripting language which is a *de facto standard* to describe particle accelerators, simulate beam dynamics, and optimise beam optics at CERN.

1.3.1 MAD-X

The current version of the software is called MAD-X [5], which was released in 2002 as the successor of MAD-8 [6]. It offers most of the MAD-8 functionalities including some additions, corrections, and extensions [7]. It was originally designed for the specific needs of the LHC [8]. The source code is mainly written in C, C++, Fortran77 and Fortran90.

Users can run MAD-X from command-line which reads and execute its scripting language. The program has a single workspace where everything is global. There is no concept of function, but a simple macro system. Deferred expressions are frequently used in many circumstances. Most of the functionalities are implemented as high-level commands customised by the user.

Almost two decades after its first release, MAD-X remains the main tool used

to describe particle accelerators inside and outside CERN. The code evolved in such a way that its maintenance and upgrades have become increasingly difficult [9]. The effort of developing new functionality can be extremely costly due to the tight coupling of its modules. Even small modifications can impact many parts of the code causing undesirable side effects. Providing new features as extensions and fixing bugs takes a significant effort in terms of time and difficulty.

The problems just mentioned imply a necessary redesign of the MAD application. Here is where a proposal for a novel version called MAD Next Generation (MAD-NG) is born.

1.3.2 MAD-NG

The Next Generation of MAD (MAD-NG) aims to leave behind the difficulties introduced by its previous version (MAD-X) over time. The project started in 2016 and it is currently released as beta version. This modern redesign of the MAD application was implemented completely from scratch with clear intentions. The main features of MAD-NG are the following: (i) simple scripting language easy to learn; (ii) modular design for scalability; (iii) it allows more flexible input; (iv) it aims to reintroduce good practices typical of software engineering in order to facilitate maintenance and bugs fixing. The software must be at least as performing as MAD-X keeping the same logic for the high-level mode of programming. It must be released as a standalone single binary file that works without any installation procedure for multiple operating systems (Linux, Windows and MacOS).

1.4 Software design decision: LuaJIT

After two years of benchmarking available cutting-edge technologies, the Lua [10] programming language has been selected for the MAD-NG application. It was accomplished a feasibility study, which proved that the proposed solution was

practical and it met the software requirements. Lua is used in a version that included LuaJIT [11], a trace-based just-in-time (JIT) compiler for the Lua programming language. MAD-NG is implemented in Lua, but it also embeds C code through the extremely efficient foreign function interface (FFI) of LuaJIT. The scripting language used by the final user of MAD-NG is also Lua, extended with some features that we needed to match the specific requirements of the application. LuaJIT was patched in order to support these extensions of syntax and semantics (e.g. lambda functions). These modifications impacted slightly the lexer, parser, bytecode and intermediate representation (IR).

The main reasons that lead to choose LuaJIT as technology are the following: (i) Lua is a general purpose language simple to use and to learn; (ii) LuaJIT shows very high performance in speed, since it is proved to be one of the most performing tracing JIT compiler ever made; (iii) it was relatively easy to embed LuaJIT into the MAD application; (iv) it is possible to extend the language for specific needs of the user; (v) LuaJIT provides an incredibly fast FFI that allows to inject C code into Lua in a very efficient way.

Before to select LuaJIT, other dynamic programming language and just-in-time compilers were taken into consideration, such as LLVM [12] and PyPy's tracing JIT compiler [13] for Python. However, LuaJIT was preferred for the motivations previously described.

A first study presented in [14] has analysed the performance of LuaJIT embedded in MAD-NG.

1.5 Thesis statement and overview

This thesis investigates with a structured and precise analysis the behaviour of programs execution during trace-based just-in-time compilation. The research was carried out in the context of the tracing just-in-time compiler LuaJIT embed-

ded in the Methodical Accelerator Design software (MAD-NG). This work was remarkably difficult due to the lack of documentation on LuaJIT, which includes sophisticated techniques and complex source code. To accomplish this research we proposed some extensions of the existing analysis tools provided by LuaJIT and we implemented some script to inspect compiler behaviour after executing a program. We aimed to fully understand how trace-based just-in-time compilation works, in order to be more conscious while taking high-level software decision and to produce JIT-friendly code.

The following Chapter 2 discusses some background on trace-based just-in-time compilation. Then, Chapter 3 gives an overview of related works and it introduces LuaJIT. Chapter 4 describes how a trace is built in LuaJIT. Chapter 5 investigates with experimental cases how multiple traces are connected to each other and organised by the compiler. Chapter 6 concludes this thesis discussing results and future works. Finally, Appendix A illustrates the analysis tools extended and implemented to study trace-based just-in-time compilation in LuaJIT.

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 [15, 16]: (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 [16] illustrates two general rules to consider when tackling the problem of compiler optimisation: (i) the *Pareto principle* (or *80/20 rule*) [17] 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* [18] 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 context 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 adequate. This choice will also decrease memory usage minimising the total amount of compiled code.

Schilling in [19] 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 [20] 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 [21], 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 [22] 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.

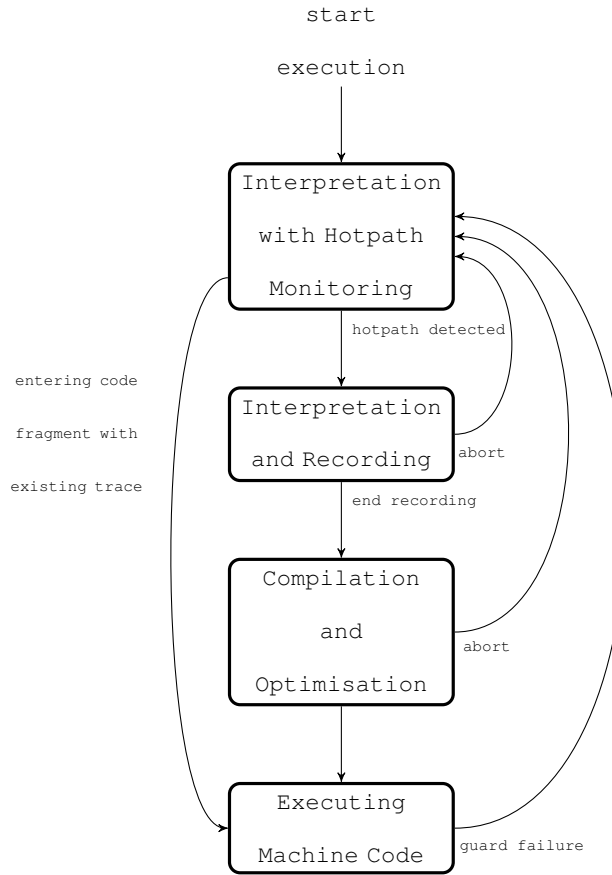


Figure 2.1: Stages of execution for a VM with tracing JIT

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 [19] 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 [23, 24], 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 [25] and PyPy's Tracing JIT Compiler [13].
- (ii). *Last Executed Iteration (LEI)*. This method, introduced by Hiniker, Hazelwood, and Smith in [26], 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 [11] 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.

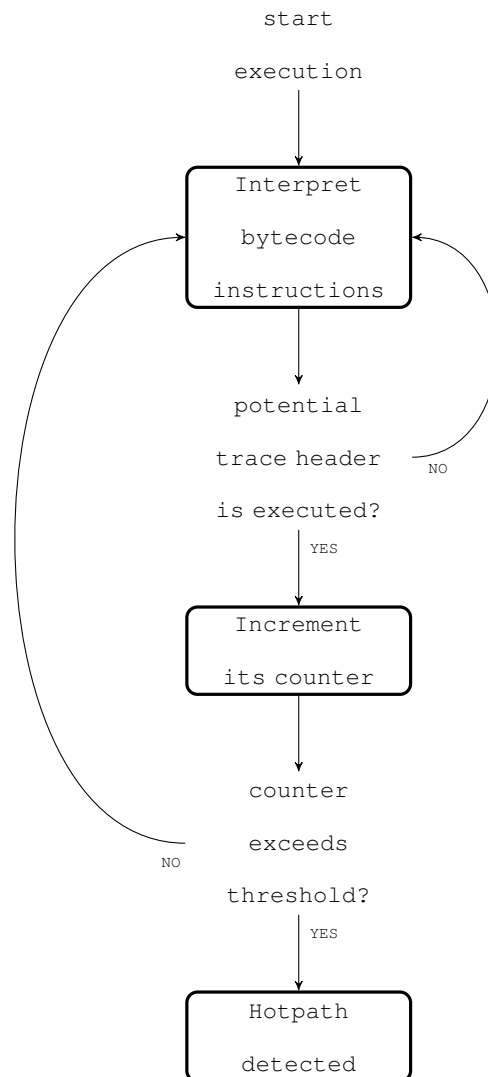


Figure 2.2: Hotpath detection

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.

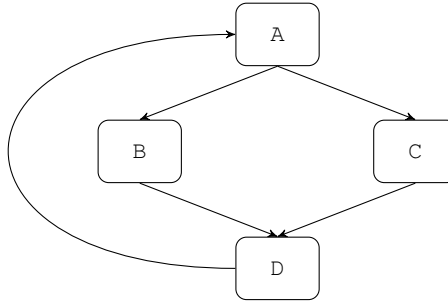


Figure 2.3: Example of loop

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 [19]: (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. [27], 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 [11]) adopt the policy that a blacklisted fragment of code cannot be whitelisted ever again. While other implementations (e.g. RaptorJIT [28]) 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.

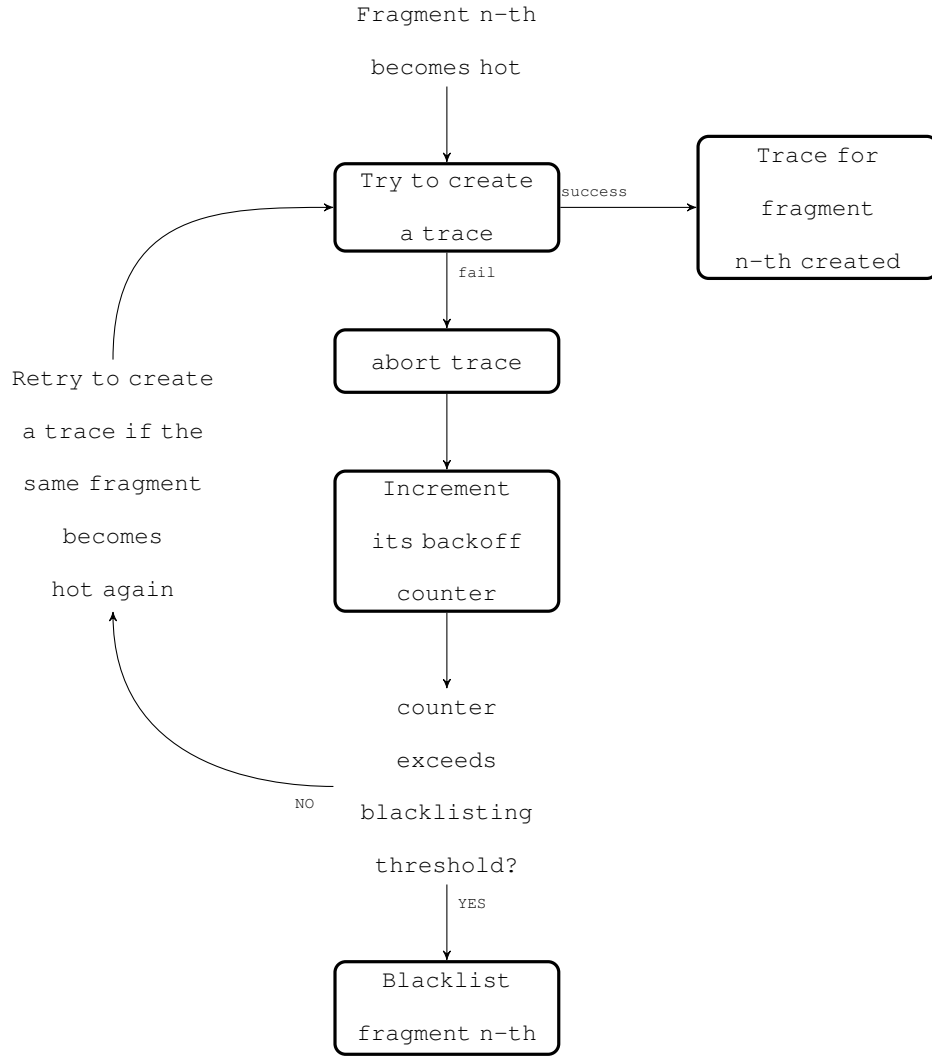


Figure 2.4: Blacklisting

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 [29]. Each variable is assigned exactly once, and every variable is defined before it is used. Gal et al. introduced in [25] 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 [23] 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 [19].

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 [11] 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.

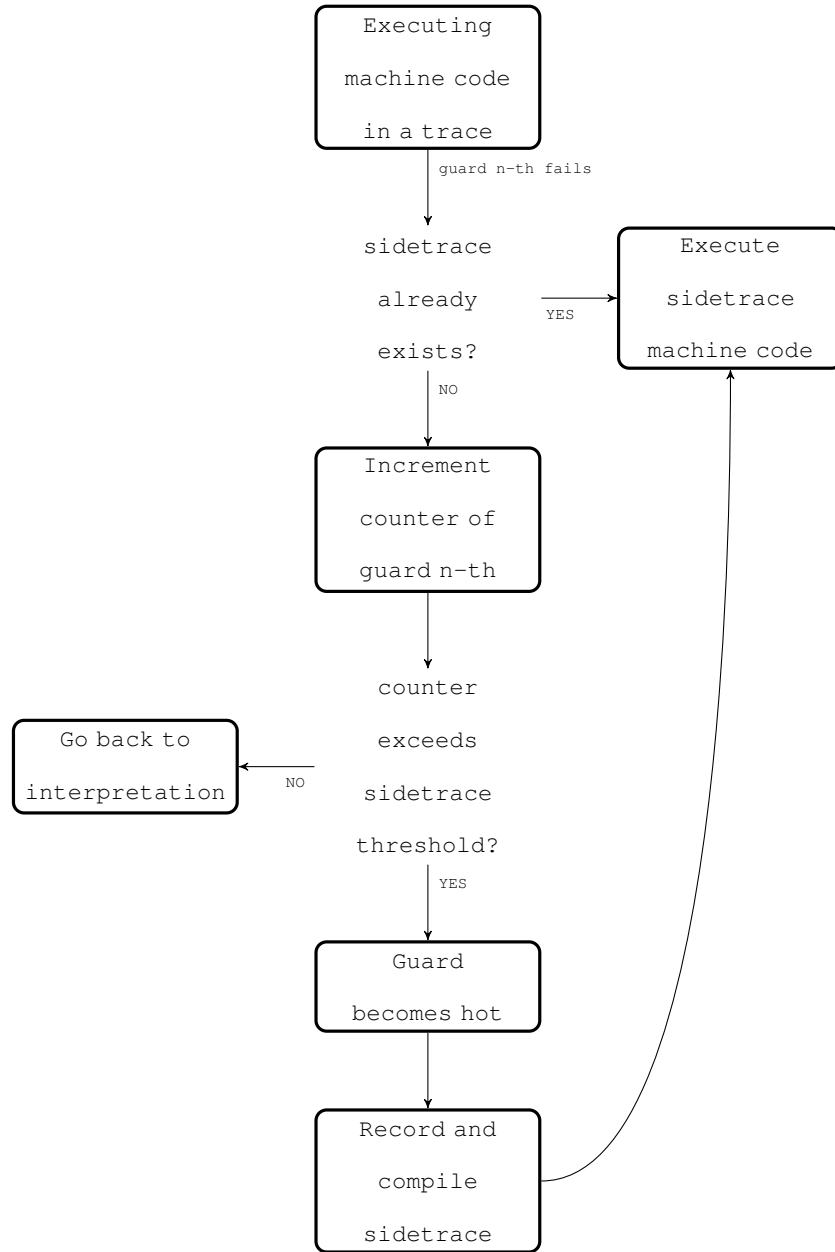


Figure 2.5: Sidetrace creation

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.

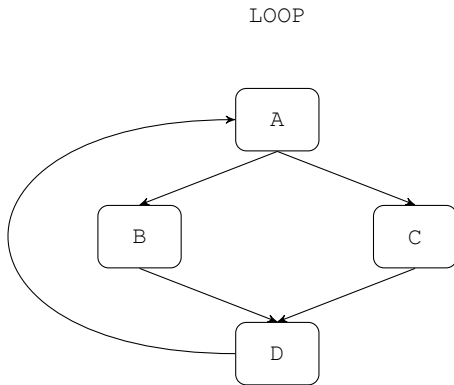


Figure 2.6: Loop

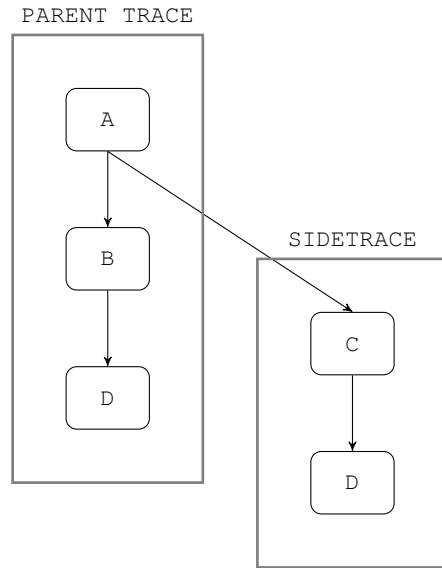


Figure 2.7: Traces generated

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 [30], 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 [11], 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.

Chapter 3

Related works

3.1 Early Tracing JITs

The idea of tracing just-in-time compilation was first introduced in 1970 by Mitchell [31]. 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 [32] 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 [33], 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.

[34] in 1999, Gschwind et al. [35] in 2000, Zheng et al. [36] 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. [37]. 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.

3.2 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. [25] 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 [29] 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 com-

pilation of dynamically-typed languages. They developed a tracing JIT compiler for JavaScript, so-called TraceMonkey [27]. It was implemented for an existing JavaScript interpreter called SpiderMonkey [38] 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 [39] released a tracing JIT called Tamarin-Tracing in 2009. Tamarin is the Adobe's VM that implements ActionScript 3 [40], a flavour of ECMAScript [41]. 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 [42] in 2006; Gal et al. [30, 43] respectively in 2006, 2007; Chang et.al [44, 45] 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. [13] 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 [46, 47, 48] respectively in 2010, 2013, 2014; Bolz's PhD thesis [49] in 2012 ; Cuni's PhD thesis [16] in 2010; Ardö et al. [50] in 2012; Vandercammen's MSc thesis [51] in 2015. On the same path Bauman et al. [52] 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. [53] 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 [11]. Its first version, LuaJIT 1, was released in 2005 provided with a JIT implemented in the assembly language DynASM [54, 55]. 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 [56]. 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 [19] for his PhD thesis in 2013. Another just-in-time compiler that is born from LuaJIT is RaptorJIT [28] 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 [57]. Finally, another relevant software that should be mentioned in this context is OpenResty [58]. It is a full-fledged web platform that integrates a modified version of LuaJIT.

3.3 An introduction to LuaJIT

3.3.1 Lua

As described in the official website, Lua [10] is 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 of the fastest interpreted scripting languages ever made.

3.3.2 LuaJIT overview

As previously explained LuaJIT [11] is a trace-based just-in-time compiler for the Lua programming 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. 3.1.

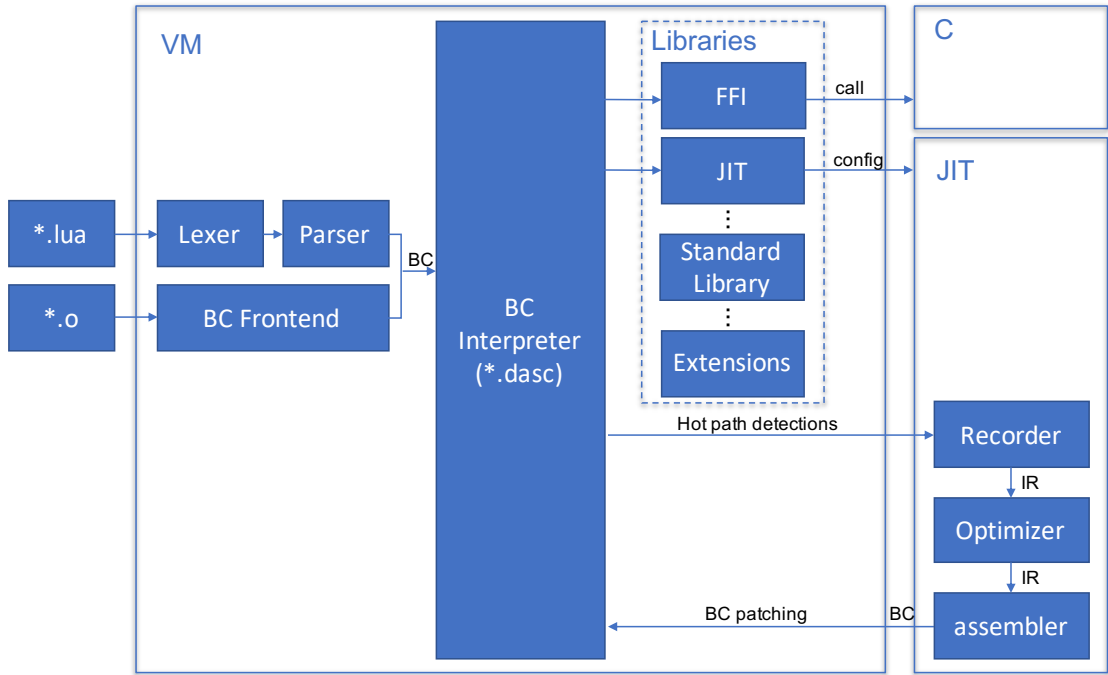


Figure 3.1: Schematic view of LuaJIT internals (source [14])

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* which 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. As previously mentioned, the interpreter of LuaJIT is fully written in assembly and it performs very fast interpretation.

Several libraries are used to support the *BC interpreter* in its tasks. Within these library the most relevant 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 which 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 *Optimizer* 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.

Chapter 4

Building a trace in LuaJIT

Traces generation in LuaJIT follows the canonical working principles characteristic of tracing JITs:

- 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 black-listed. 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.

```
1 typedef uint16_t HotCount;
```

```
2
```

```

3 /* Number of hot counter hash table entries (must be a power of two). */
4 #define HOTCOUNT_SIZE    64
5
6 /* Global state */
7 typedef struct GG_State {
8     ...
9     HotCount hotcount[HOTCOUNT_SIZE]; /* Hot counters. */
10    ...
11 } 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.

```

1 /* Hotcount decrements. */
2 #define HOTCOUNT_LOOP    2
3 #define HOTCOUNT_CALL    1
4
5 #define JIT_PARAMDEF(_) \
6     _(\007, hotloop, 56) /* # of iter. to detect a hotloop/call. */ \
7
8 /* Initialize hotcount table. */
9 void lj_dispatch_init_hotcount(global_State *g)
10 {
11     int32_t hotloop = G2J(g)->param[JIT_P_hotloop]; /* extract hotloop value = 56 */
12     HotCount start = (HotCount)(hotloop*HOTCOUNT_LOOP - 1); /* start = 111 */

```

```

13  HotCount *hotcount = G2GG(g)->hotcount;
14  uint32_t i;
15  for (i = 0; i < HOTCOUNT_SIZE; i++)
16    hotcount[i] = start; /* init hotcounters to 111 */
17 }

```

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`.

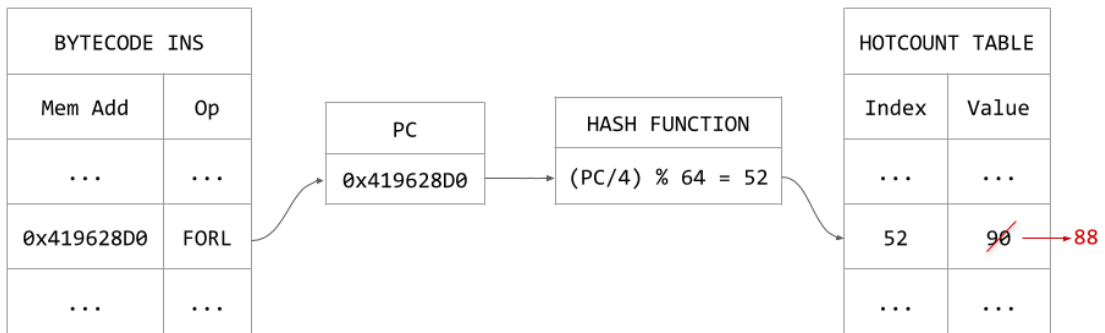


Figure 4.1: Hotcount decrement for loops

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

```

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

```

```
4 (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. [25], 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 ($0 \times 1n$) or *not Active* if the activation bit is set to 0 ($0 \times 0n$). When an error occurs the trace compiler changes its current state to *not active* before switching to state `LJ_TRACE_ERR` and abort.

```
1 /* Trace compiler state. */
2 typedef enum {
3     LJ_TRACE_IDLE, /* Trace compiler idle */
4     LJ_TRACE_ACTIVE = 0x10,
5     LJ_TRACE_RECORD, /* BC recording active */
6     LJ_TRACE_START, /* New trace started */
7     LJ_TRACE_END, /* End of trace. */
8     LJ_TRACE_ASM, /* Assemble trace */
9     LJ_TRACE_ERR /* Trace aborted with error */
10 } 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`.

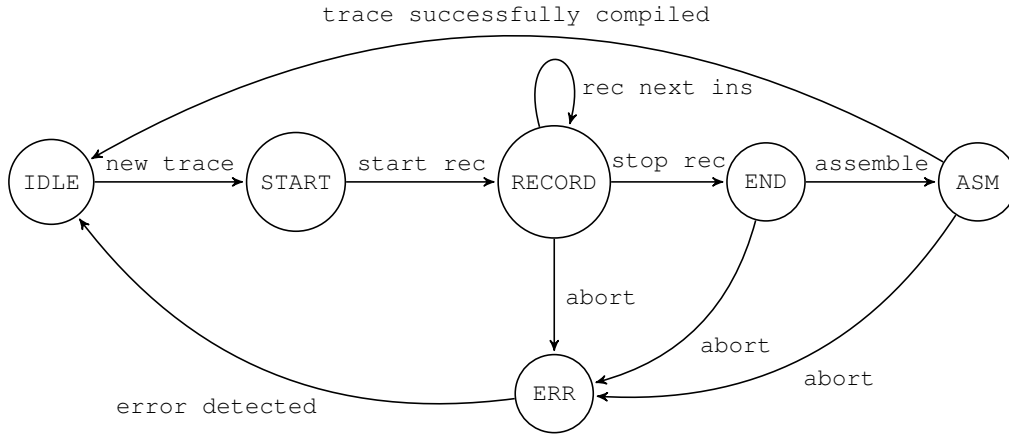


Figure 4.2: Trace compiler state machine

```

1  /* State machine for the trace compiler*/
2  static TValue *trace_state(...) {
3      do {
4          retry:
5          switch (J->state) {
6              case LJ_TRACE_START:
7                  J->state = LJ_TRACE_RECORD;
8                  trace_start(J);
9                  lj_dispatch_update(J2G(J));
10                 break;
11
12             case LJ_TRACE_RECORD:
13                 setvmstate(J2G(J), RECORD);
14                 lj_vmevent_send(L, RECORD, ...);
15                 lj_record_ins(J);
16                 break;
17
18             case LJ_TRACE_END:
19                 setvmstate(J2G(J), OPT);
20                 /* Perform optimisations */
21                 lj_opt_dce(J);
22                 /* Loop optimisation failed? */
23                 if (lj_opt_loop(J)) {
24                     ...
25
26                 /* Try to continue recording*/
27                 J->state = LJ_TRACE_RECORD;
28                 break;
29
30                 lj_opt_split(J);
31                 lj_opt_sink(J);
32                 J->state = LJ_TRACE_ASM;
33                 break;
34
35             case LJ_TRACE_ASM:
36                 setvmstate(J2G(J), ASM);
37                 lj_asm_trace(J, &J->cur);
38                 trace_stop(J);
39                 setvmstate(J2G(J), INTERP);
40                 J->state = LJ_TRACE_IDLE;
41                 lj_dispatch_update(J2G(J));
42                 return NULL;
43
44             default:
45                 /* Trace aborted asynchronously*/
46                 setintv(L->top++, LJ_TRError_RECERR);
47                 /* fallthrough */
48
49             case LJ_TRACE_ERR:

```

```
49     if (trace_abort(J)) {
50         goto retry;
51     }
52     setvmstate(J2G(J), INTERP);
53     J->state = LJ_TRACE_IDLE;
54     lj_dispatch_update(J2G(J));
55     return NULL;
56 }
57
58 } while (J->state > LJ_TRACE_RECORD);
59 return NULL;
60 }
```

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 [59]. Finally, the trace compiler state is changed to `LJ_TRACE_ASM`.

4.3 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. 4.4).

```

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 4.4: Bytecode instructions to force JIT-compiled trace execution

4.4 Abort

An abort can occur at any stage of the just-in-time compilation: recording, optimisation or assembling. Tab. 4.5 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 4.5: 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

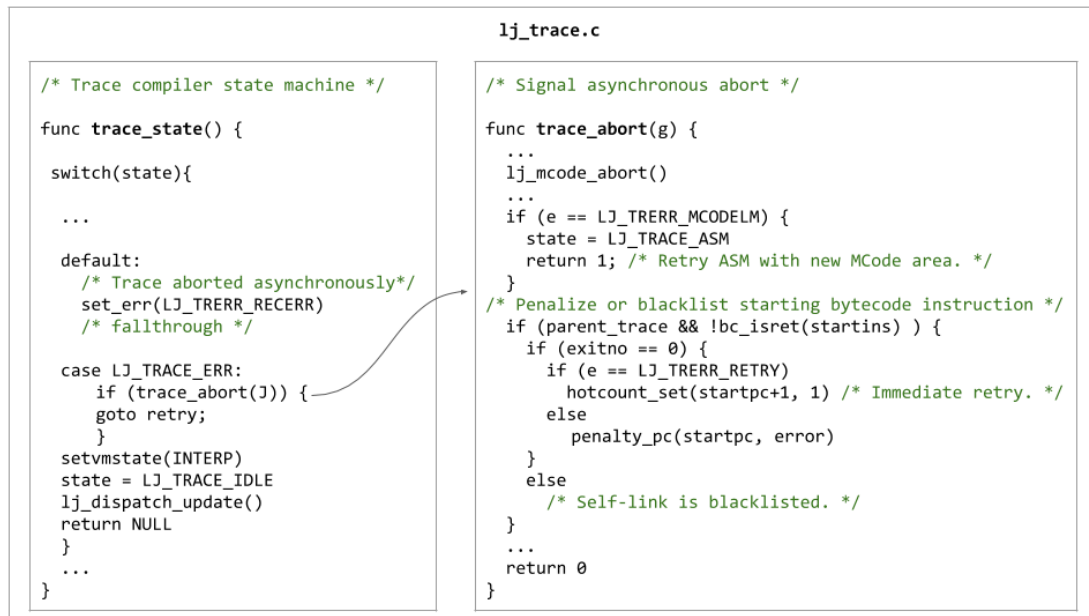


Figure 4.8: Abort

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. 4.6

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 4.6: Parameters of the JIT compiler

4.5 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. [27], 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 4.7).

'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 4.7: 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. 4.5).

```

1 /* Round-robin penalty cache for bytecodes leading to aborted traces. */
2 typedef struct HotPenalty {
3     MRef pc;      /* Starting bytecode PC. */
4     uint16_t val; /* Penalty value, i.e. hotcount start. */
5     uint16_t reason; /* Abort reason (really TraceErr). */
6 } HotPenalty;
7
8 #define PENALTY_SLOTS 64 /* Penalty cache slot. Must be a power of 2. */
9 #define PENALTY_MIN (36*2) /* Minimum penalty value. */
10 #define PENALTY_MAX 60000 /* Maximum penalty value. */
11 #define PENALTY_RNDBITS 4 /* # of random bits to add to penalty value. */
12
13 /* JIT compiler state. */
14 typedef struct jit_State {
15     ...

```

```

16 HotPenalty penalty[PENALTY_SLOTS]; /* Penalty slots. */
17 uint32_t penaltyslot; /* Round-robin index into penalty slots. */
18 uint32_t prngstate; /* PRNG state. */
19 ...
20 } 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. 4.8 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 4.8: 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 `IL...` 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.

```

1  /* Penalize a bytecode instruction. */
2  static void penalty_pc(jit_State *J, GCproto *pt, BCIns *pc, TraceError e) {
3      uint32_t i, val = PENALTY_MIN;
4      for (i = 0; i < PENALTY_SLOTS; i++)
5          if (mref(J->penalty[i].pc, const BCIns) == pc) { /* Cache slot found? */
6              /* First try to bump its hotcount several times. */
7              val = ((uint32_t)J->penalty[i].val << 1) + LJ_PRNG_BITS(J, PENALTY_RNDBITS);
8              if (val > PENALTY_MAX) {
9                  blacklist_pc(pt, pc); /* Blacklist it, if that didn't help. */
10                 return;
11             }
12             goto setpenalty;
13         }
14     /* Assign a new penalty cache slot. */
15     i = J->penaltyslot;
16     J->penaltyslot = (J->penaltyslot + 1) & (PENALTY_SLOTS-1);
17     setmref(J->penalty[i].pc, pc);
18     setpenalty:
19     J->penalty[i].val = (uint16_t)val;
20     J->penalty[i].reason = e;
21     hotcount_set(J2GG(J), pc+1, val);
22 }
23
24 /* Blacklist a bytecode instruction. */
25 static void blacklist_pc(GCproto *pt, BCIns *pc) {
26     setbc_op(pc, (int)bc_op(*pc) + (int)BC_ILOOP - (int)BC_LOOP);
27     pt->flags |= PROTO_ILOOP;
28 }

```

Listing 4.8: lj_trace.c

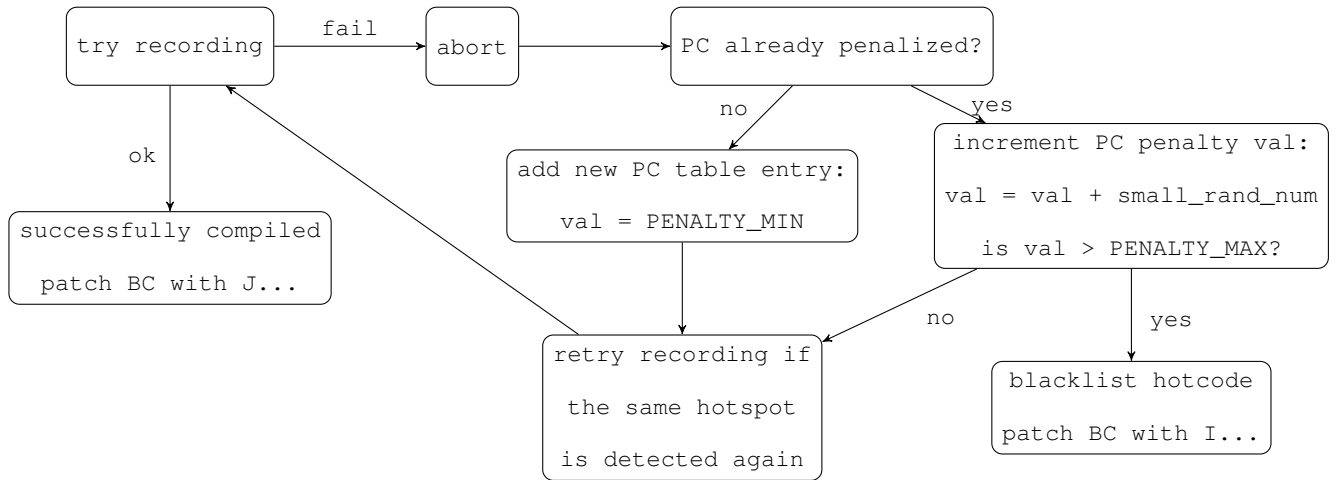


Figure 4.9: Penalization and blacklisting

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.

4.5.1 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 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 byte-code instruction (`1...`) 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 in 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 community². This example is proposed again below to better illustrate the problem.

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

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

```
1  -- Blacklisting Cancer
2
3  -- Call a function with an argument.
4  local function apply (f, x)
5      return f(x)
6  end
7
8  -- Return the value x wrapped inside a closure.
9  local function fwrap (x)
10     return function () return x end -- Create a closure Not-Yet-Implemented (NYI)
11 end
12
13 -- Attempt to compile that naturally fails.
14 for i=1,1e5 do
15     apply(fwrap, i) -- Abort message: NYI: create a closure
16 end
17
18 -- apply() function is now permanently blacklisted.
19 -- Every call to apply() in the future will have to be run by the interpreter.
20
21 local function nop (x)
22 end
23
24 -- Attempt to compile that should not fail.
25 for i=1,1e5 do
26     apply(nop, i) -- Abort message: calling blacklisted function
27 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 [28] 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.

Chapter 5

LuaJIT traces investigation

5.1 Introduction

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.

The first section clarifies how the just-in-time compiler (JIT) generate traces from simple loops. The second and third sections show how traces are connected with each other in more complex structures. Finally, the last section 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.

5.2 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 pre-conditions 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 [60]. 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.

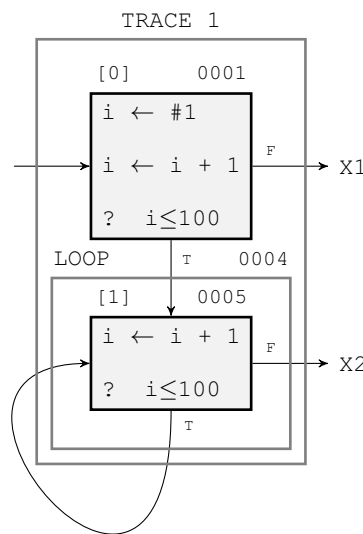


Figure 5.1: Trace flow diagram empty loop

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 loops¹ 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  -- 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

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

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	0007 > int LE 0006 0001
---- TRACE 1 IR	0008 ----- LOOP -----
0001 > int SLOAD #5 CRI	0009 + num ADD 0005 +22
0002 > int LE 0001 +2147483646	0010 + int ADD 0006 +1
0003 int SLOAD #4 CI	0011 > int LE 0010 0001
0004 > num SLOAD #1 T	0012 int PHI 0006 0010
0005 + num ADD 0004 +22	0013 num PHI 0005 0009
0006 + int ADD 0003 +1	---- 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 SLOADs 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.

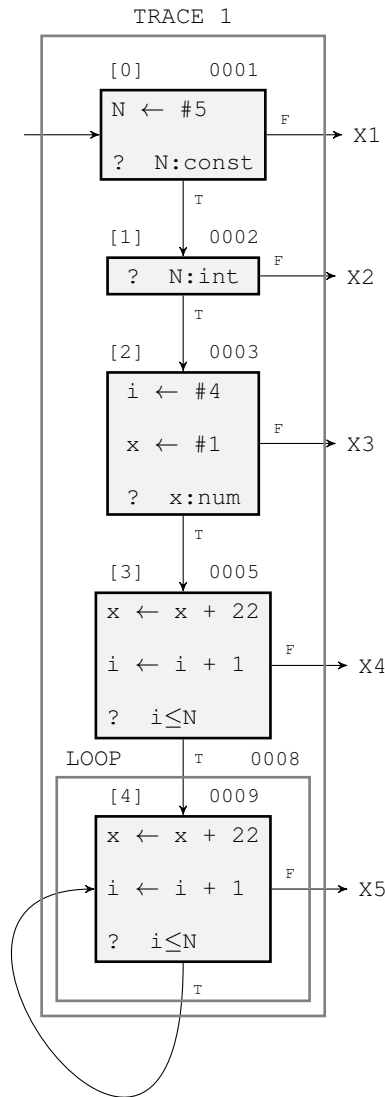


Figure 5.2: Trace flow diagram loop with assignment

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	---- TRACE 1 stop -> loop
0006 ADDVN 0 0 0 ; 11	
0007 MODVN 5 4 1 ; 10	---- TRACE 2 start 1/4 Ex.lua:8
0008 ISNEN 5 2 ; 0	0010 ADDVN 0 0 3 ; 22
0009 JMP 5 => 0011	0011 ADDVN 0 0 4 ; 33
0011 ADDVN 0 0 4 ; 33	0012 JFORL 1 1
0012 FORL 1 => 0006	---- 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.

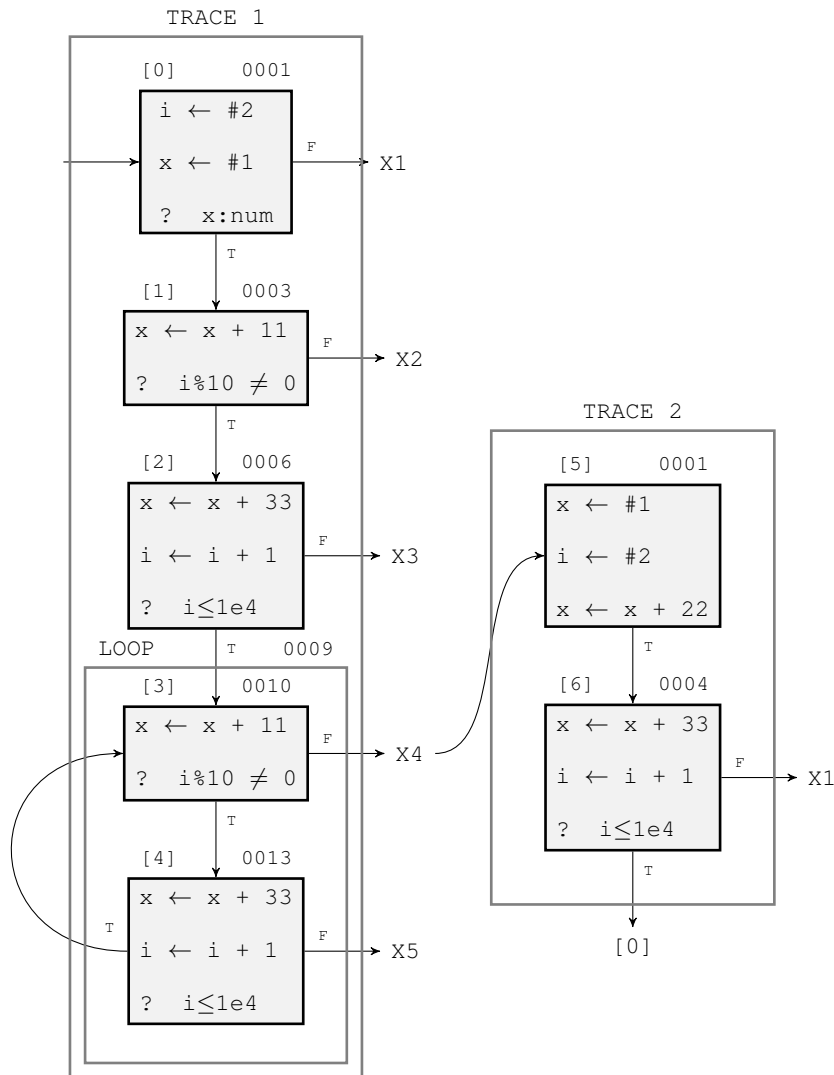


Figure 5.3: Trace flow diagram loop with if-statement

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	0013	ADDVN	0	0	2	; 33
0011	ADDVN	0	0	1			; 22
0012	FORL	5	=>	0011			
----	TRACE 1 stop -> loop	0007	KSHORT	5	2		
		0008	KSHORT	6	1000		
----	TRACE 2 start 1/3 Ex.lua:10	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
```

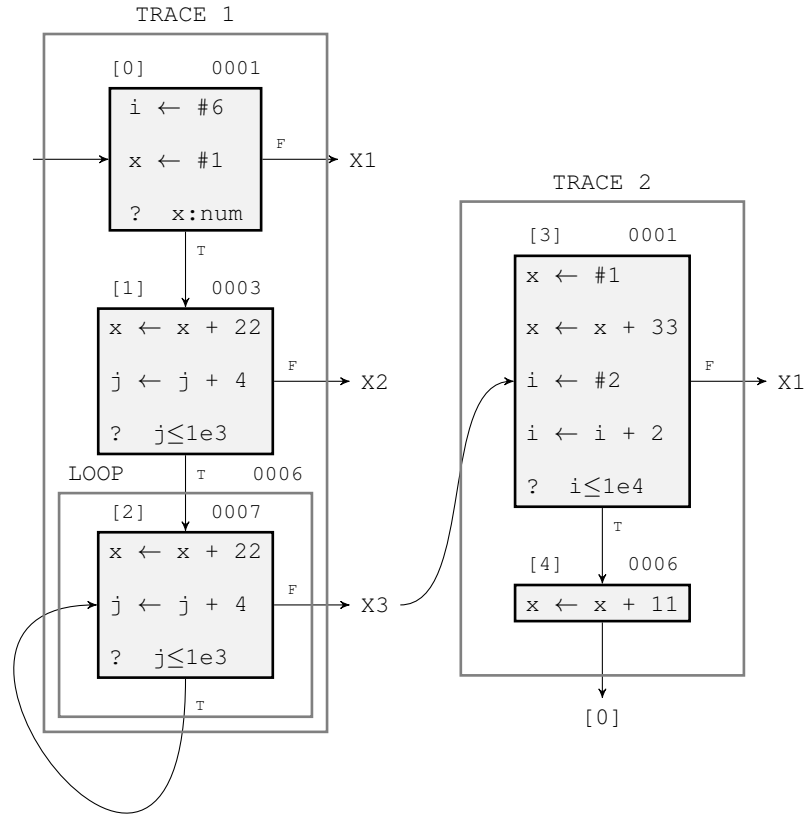


Figure 5.4: Trace flow diagram nested loop

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 inlined².

²Conversation 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.

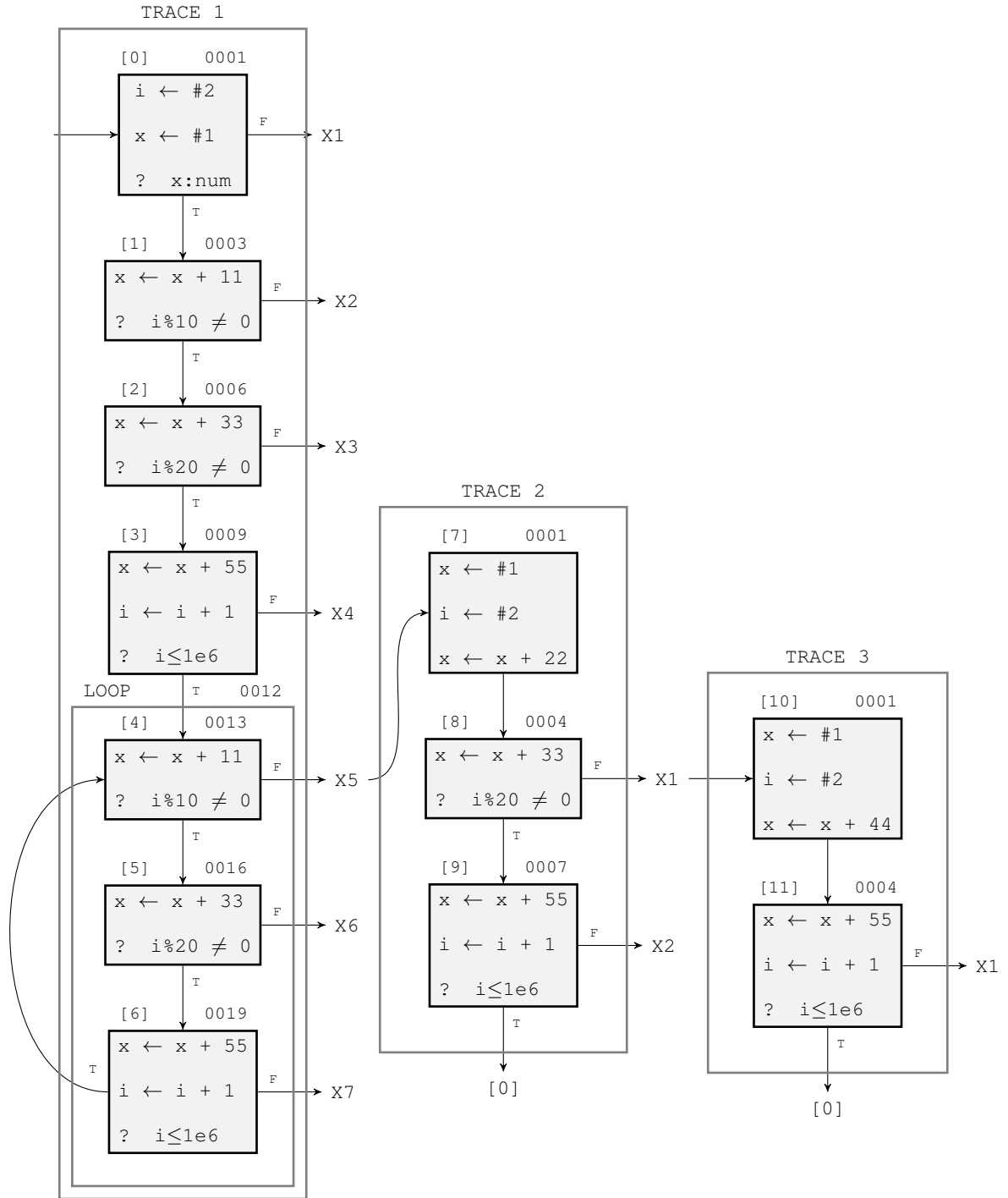


Figure 5.5: Trace flow diagram loop with 2 if-statements Example 1

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      0002 int SLOAD #2 PI
0011 num CONV     0009 num.int      0003 num ADD 0001 +44
---- TRACE 2 stop -> 1      0004 num ADD 0003 +55
                             0005 int ADD 0002 +1
---- TRACE 3 start 1/6 Ex.lua:12  0006 > int LE 0005 +1000000
---- TRACE 3 IR      0007 num CONV 0005 num.int
0001 num SLOAD #1 PI      ---- 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.

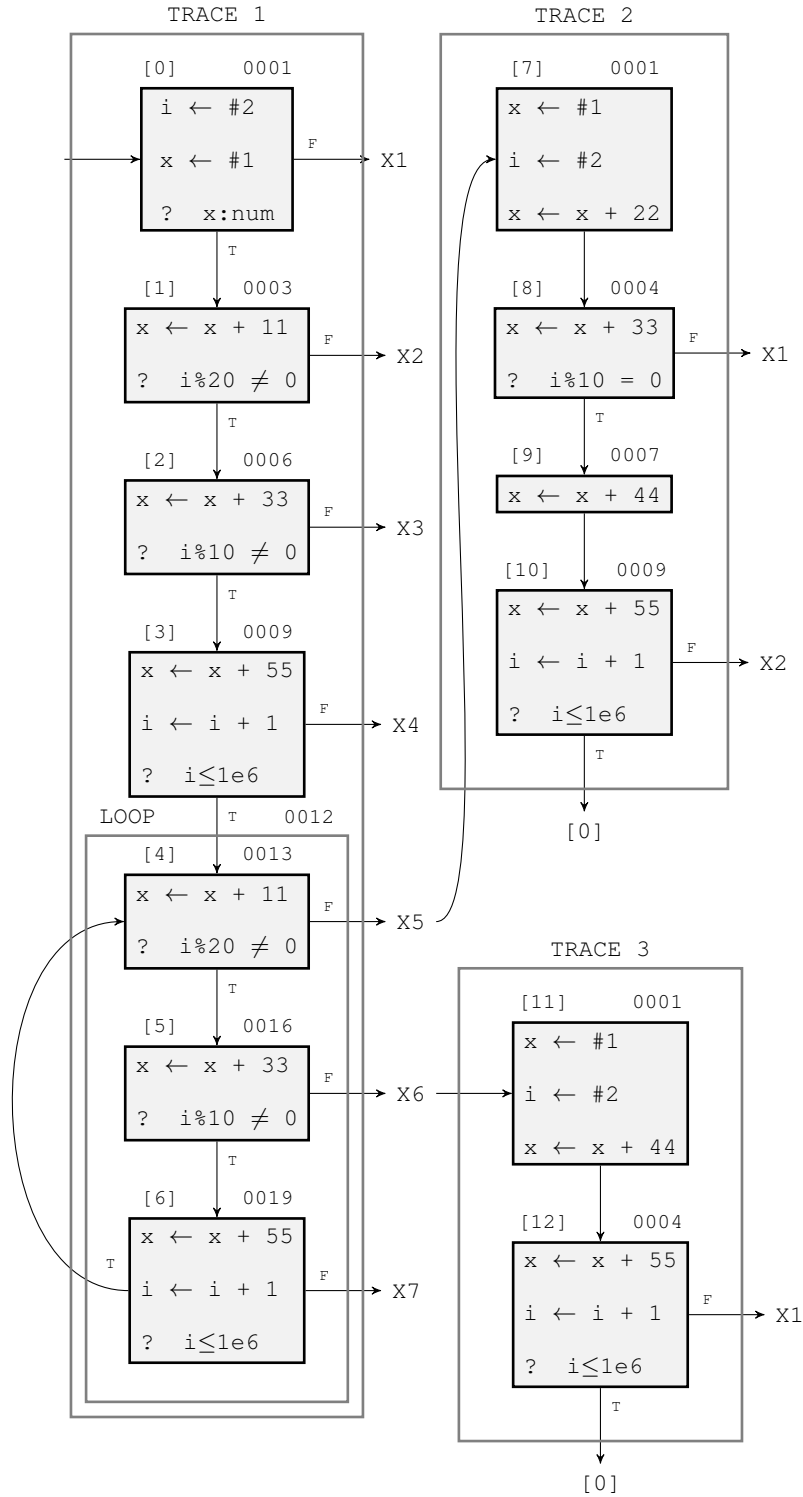


Figure 5.6: Trace flow diagram loop with 2 if-statements Example 2

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	0011	ADDVN	0	0	5	; 33					
0006	ADDVN	0	0	1		; 11	0012	MODVN	5	4	2	; 10
0007	MODVN	5	4	2		; 10	0013	ISNEN	5	3		; 0
0008	ISNEN	5	3			; 0	0014	JMP	5 =>	0016		
0009	JMP	5 =>	0011				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

```

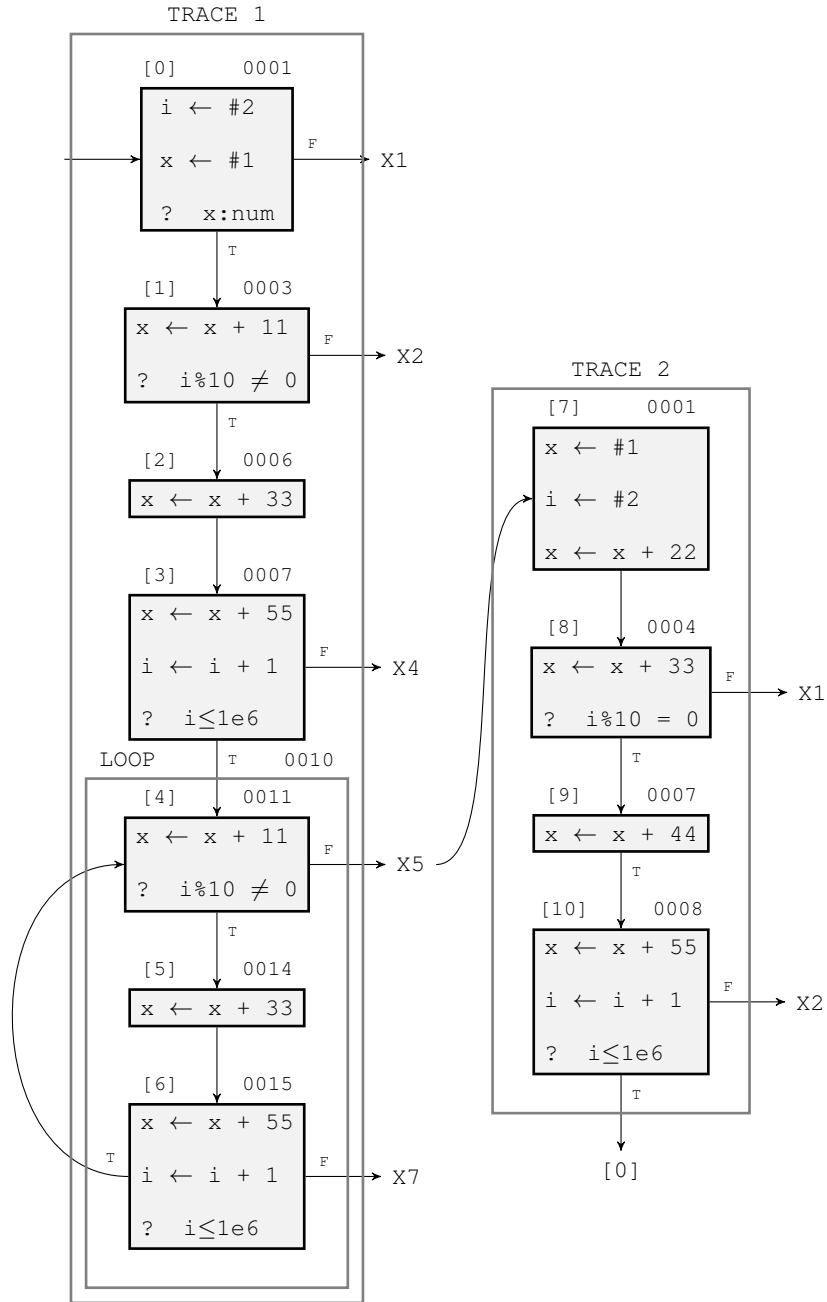


Figure 5.7: Trace flow diagram loop with 2 if-statements Example 3

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. $i = 60, 120, \dots$).

```

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

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 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.

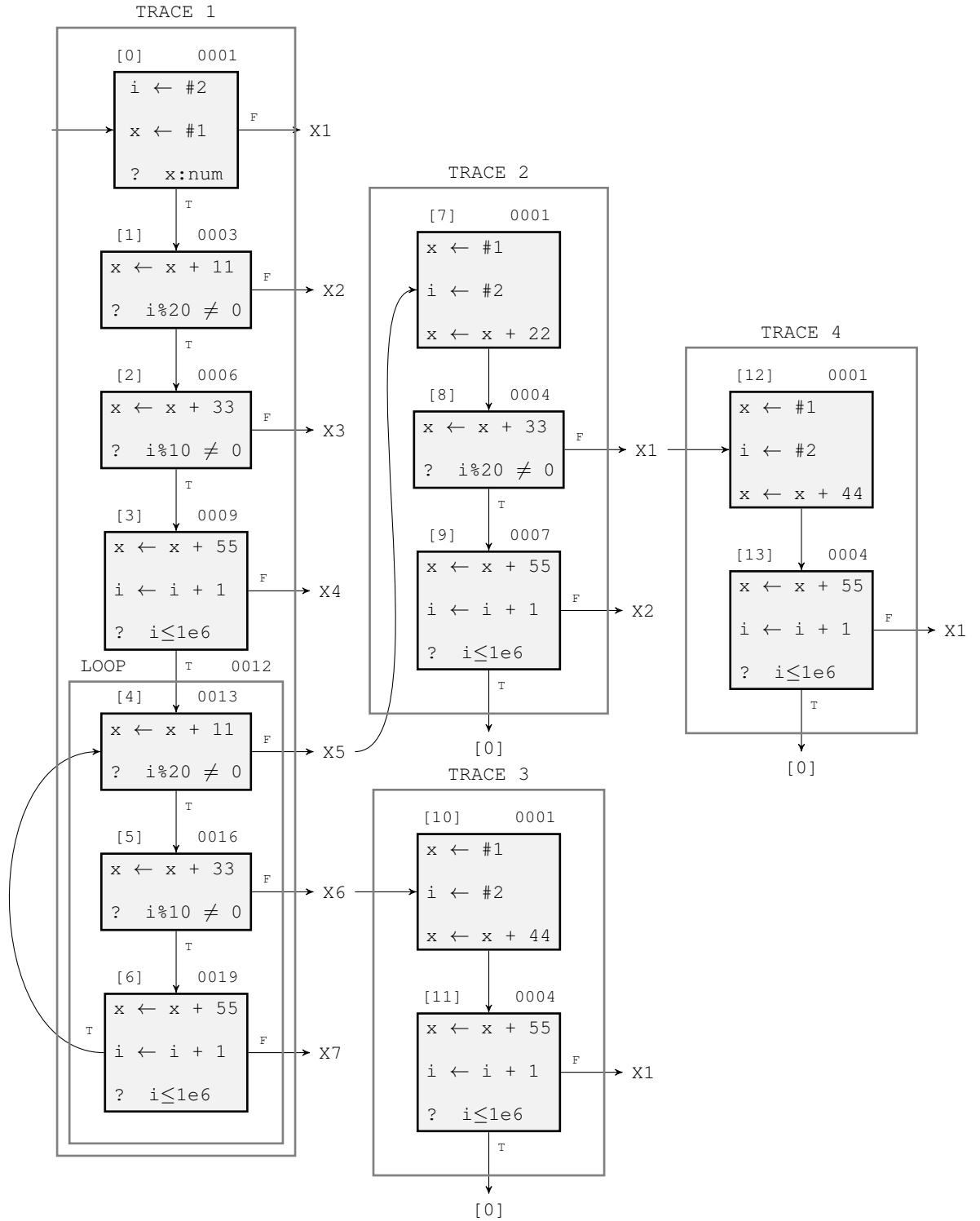


Figure 5.8: Trace flow diagram loop with 2 if-statements Example 4

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                                ---- TRACE 1 stop -> loop
0011 ADDVN      0    0    1 ; 22
0012 FORL       5 => 0011                                ---- TRACE 2 start *.lua:11

```

```

0018  ADDVN    0  0  3  ; 44
0019  FORL     5 => 0018          ---- TRACE 4 start 2/3 *.lua:14
---- TRACE 2 stop -> loop      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

---- 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

```

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

```

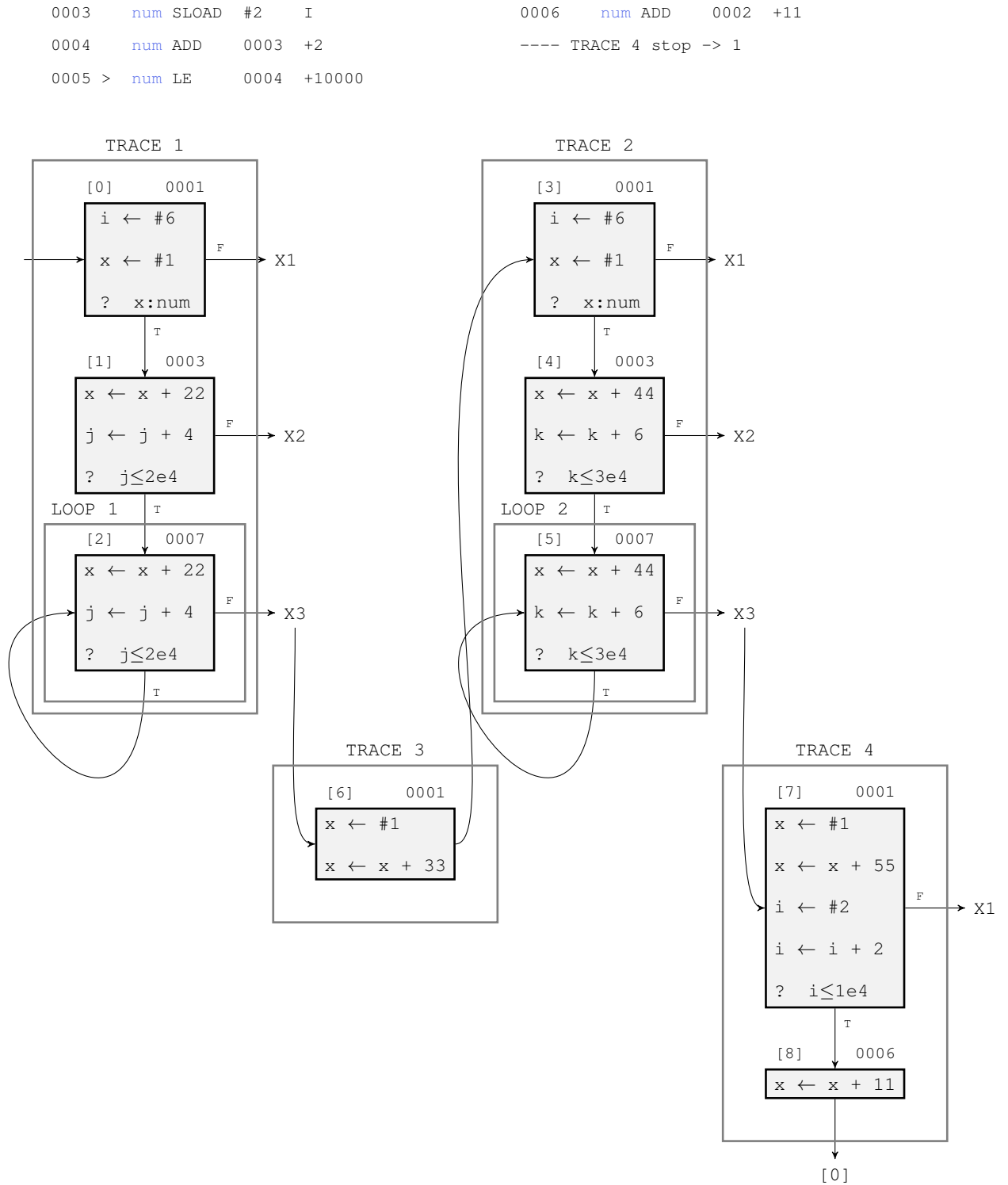


Figure 5.9: Trace flow diagram loop with 2 inner loops

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
```

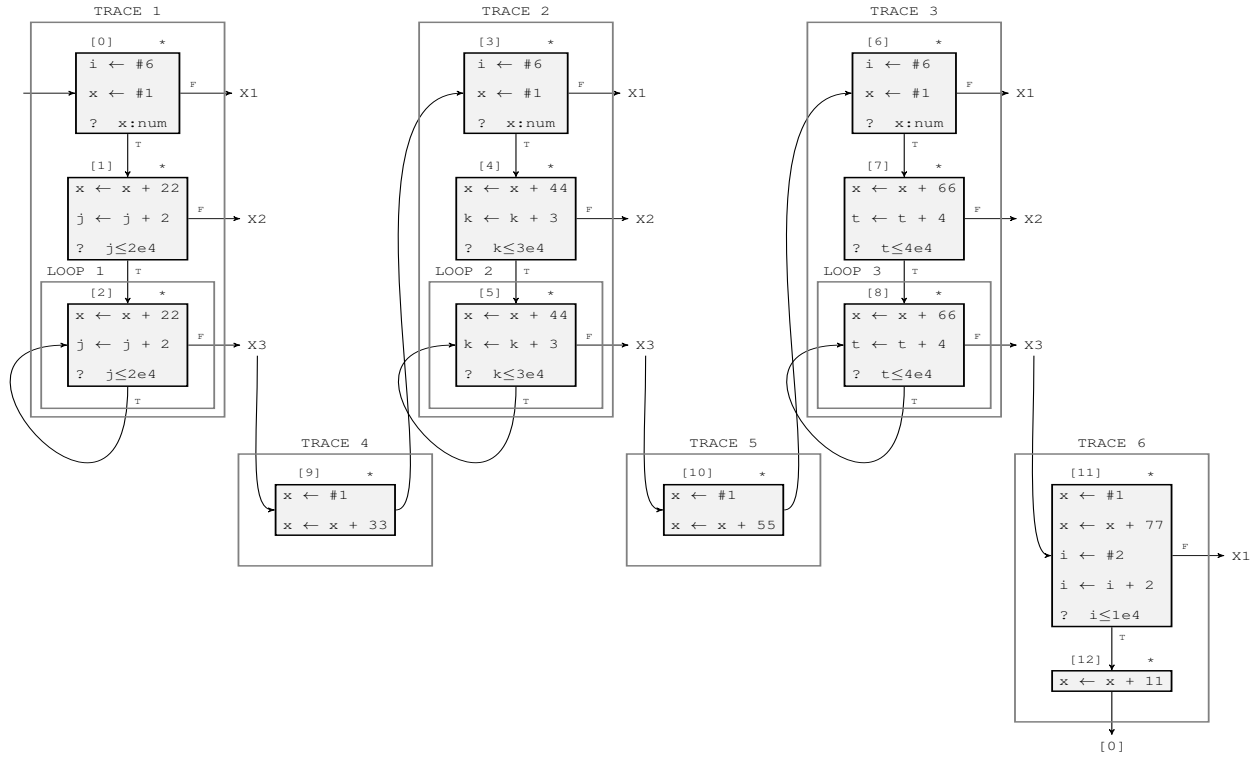


Figure 5.10: Trace flow diagram loop with 3 inner loops

5.5 Function

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

Chapter 6

Conclusions

6.1 Overview

This thesis investigated trace-based just-in-time (JIT) compilation applied to the context of the Next Generation of the Methodical Accelerator Design software (MAD-NG) [4].

Trace-based just in time compilation is a technique used for dynamic interpreted languages. It consists of recording a linear path of frequently executed operations, so-called *trace*, compiling it to native machine code and executing it.

MAD-NG provides modules to describe particle accelerators, simulate beam dynamics, and optimise beam optics. It has to process a massive quantity of data and operations with user-defined expressions during simulation, hence it needs to be very efficient. A technology that fulfils these demands is LuaJIT [11], a trace-based just-in-time compiler for the Lua [10] programming 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. LuaJIT was embedded into MAD application in order to achieve high performances thanks to its tracing JIT.

This thesis accomplished an extensive and precise study on the behavioural

analysis of programs execution during trace-based just-in-time compilation. LuaJIT cannot be used as a blackbox, it was necessary to deeply understand all the mechanisms performed by the compiler under the hood. The consciousness of the internal mechanisms behind a tracing JIT compiler is necessary in order to produce JIT-friendly code.

We proposed some extension of the existing tools provided by LuaJIT to analyse how the compiler organises traces when executing programs. In particular, we extended the Dump mode with useful information about generated traces. Moreover, we implemented some post-processing scripts to inspect the traces generated by the JIT during the execution of a program. Post-execution traces analysis aims to collect summary information about traces to get an overall picture of how they were organised by the JIT. This gives to the user the chance to adapt his code in a way that it can better profit of trace-based compilation. Moreover, this can give some insights to detect critical parts of the source code, which can impact the application performance.

6.2 Difficulties and strategy

Despite its outstanding performance, LuaJIT suffers from some issues that also affected this research work. Firstly, there is no comprehensive documentation of LuaJIT internals, but only a short summary of techniques used is given by Mike Pall (author of LuaJIT) in a public statement about its intellectual properties [56]. This is a major issue specially for entry-level users, which struggle to fully understand the rationale applied by the JIT. The wiki on the official website [59] is a thin documentation, which does not illustrate all the necessary details. Useful information are spread on various websites and mostly in the LuaJIT mailing list where users contribute to illustrate characteristics of the compiler. It can be very tricky to find specific information about the JIT, which in some cases must

be extracted directly from the code. Sometimes, the programming style of Pall can be really complex to decode, specially for the parts implemented in assembler. High execution performance is obtained at the expense of having more complicated code.

In this research work we approached LuaJIT considering it as a blackbox at first and analysing traces generated by the compiler through the Dump mode. This gave a general idea of how specific structures of the code were translated into traces by the JIT. Then, we reconstructed an overview of LuaJIT architecture through information spread on the web and related works. Finally, we aimed for a more detailed understanding decoding information from the source code. This step-by-step methodology is suggested to any newcomers interested to investigate LuaJIT internals.

Extracting the overall behaviour of LuaJIT was essential to be more conscious of what happens under the hood. This knowledge positively influenced high-level software design decisions that were taken in order to produce code with JIT-friendly style. 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.

6.3 Results

The major achievement accomplished with this thesis was to fully understand how trace-based just-in-time compilation is performed by LuaJIT in the context of the Methodical Accelerator Design software (MAD-NG). Implementing some extensions to the Dump mode, and through post-execution analysis we realised a precise and accurate investigation of trace-based just-in-time compilation presenting some experimental cases. We aimed to discover the best practices to adopt in order to profit the most from the potential of trace-based just-in-time compilation.

Moreover, the techniques applied to observe tracing JIT execution intend to be used for MAD-NG in order to detect critical parts of the application which must be redesigned in some JIT-friendly style.

We are now more conscious of what is the potential and the drawbacks of trace-based just-in-time compilation. High-level decisions on the application are taken in such a way that programs execution exploits the extremely efficient performances of trace-based just-in-time compilation.

Among the presented achievements, this thesis can be considered as a starting document that describes the details of trace-based just-in-time compilation performed by LuaJIT. It intends to be used as a source for newcomers in the LuaJIT community interested to discover how the compiler behaves, while executing programs, generating traces and connecting them into more complex structures.

6.4 Future works

This thesis was carried out in the context of a project at CERN for the Methodical Accelerator Design software (MAD) supported by the Accelerators and Beam Physics group in the Beams department. My personal prospect is to continue working on this project progressing the studies on trace-based just-in-time compilation. The intention is to continue extending the tools of analysis for the JIT compiler and designing a general strategy to use it. The parts of MAD-NG that will be revealed to be written in a non JIT-friendly style must be redesigned in order to fully profit of trace-based just-in-time compilation.

We also aims to create a public document about LuaJIT that can be modified and enriched by its community. This will promote the technology and it will give guidelines on how to use it.

Appendix A

Analysis Tools

This appendix presents some diagnostic tools for the analysis of LuaJIT. The compiler framework already provides a set of tools that helps to investigate what happens under the hood while executing a program with LuaJIT. Part of the work for this thesis was to improve the reported information provided by these tools in order to study the behaviour of LuaJIT more in details. In particular, the dump mode was extended to get more specific information on traces. In this way programmers will have some insights for writing a JIT-friendly code since in specific circumstance certain patterns of code can be preferred to others.

A.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 using this mode 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.

A.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 \geq 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 A.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()
```

A.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 A.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 [61].

```
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 A.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 [62].

```
....          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
```

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).

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' flag 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 A.4: IR instructions

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
10020ffe0 addsd xmm7, xmm6
->LOOP:
10020ffea jle 0x10020ffd0 ->LOOP
```

A.4 Dump mode extensions

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.

The first extension implemented for this thesis has introduced more information about traces status. 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.
- (ii). The second line (my 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 ($PC=0x\dots$), 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`, number of another trace, `tail-recursion`, `interpreter`, 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 Hotcount 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 Hotcount 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 Hotpenalty 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 ($\text{valpenalty} = \text{val} * 2 + \text{small_rand_num}$) exceeds the maximum penalty value of 60000 (default of LuaJIT). See 4.5 for more details on blacklisting in LuaJIT.

A.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 of this thesis, some scripts have been implemented 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.

Bibliography

- [1] CMS Collaboration. “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC”. In: *Physics Letters B* 716.1 (2012), pp. 30–61.
- [2] ATLAS Collaboration. “Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC”. In: *Physics Letters B* 716.1 (2012), pp. 1–29.
- [3] Esma Mobs. “The CERN accelerator complex. Complexe des accélérateurs du CERN”. In: (2016). General Photo.
- [4] CERN. *Methodical Accelerator Design*. URL: <http://mad.web.cern.ch/mad/>.
- [5] CERN. *MAD-X*. 2002. URL: <http://madx.web.cern.ch/madx/>.
- [6] H Grote and F Schmidt. “MAD-X-an upgrade from MAD8”. In: *Proceedings of the 2003 Particle Accelerator Conference*. Vol. 5. IEEE. 2003, pp. 3497–3499.
- [7] F Schmidt. “Mad-X a worthy successor for MAD8?” In: (2006).
- [8] CERN. *The MAD-X Program: User’s Reference Manual*. 2018. URL: <https://mad.web.cern.ch/mad/webguide/manual.html>.
- [9] L Deniau. “MAD-X progress and future plans”. In: CERN-ATS-Note-2012-078 MD (2012), 16 p.

- [10] PUC-Rio Tecgraf. *The Programming Language Lua*. URL: <http://www.lua.org/>.
- [11] Mike Pall. *LuaJIT 2*. 2012. URL: <http://luajit.org>.
- [12] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for life-long program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and run-time optimization*. IEEE Computer Society. 2004, p. 75.
- [13] 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.
- [14] Aurelien Bloch. “First Performance Analysis of MAD-NG Embedded Tracing JIT Compiler”. In: (2018).
- [15] Matthew Arnold et al. “A survey of adaptive optimization in virtual machines”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 449–466.
- [16] 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.
- [17] Wikipedia contributors. *Pareto principle — Wikipedia, The Free Encyclopedia*. 2019.
- [18] 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.
- [19] Thomas Schilling. “Trace-based Just-in-time Compilation for Lazy Functional Programming Languages”. PhD thesis. University of Kent, 2013.

- [20] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005. Chap. 2.
- [21] 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.
- [22] 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.
- [23] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. *Transparent dynamic optimization: The design and implementation of Dynamo*. Tech. rep. HP Laboratories Cambridge, 1999.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] Andreas Gal et al. “Trace-based just-in-time type specialization for dynamic languages”. In: *ACM Sigplan Notices* 44.6 (2009), pp. 465–478.

- [28] Luke Gorrie. *RaptorJIT*. 2017. URL: <https://github.com/raptorjit/raptorjit>.
- [29] 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.
- [30] Andreas Gal and Michael Franz. *Incremental dynamic code generation with trace trees*. Tech. rep. Citeseer, 2006.
- [31] James G Mitchell. “The design and construction of flexible and efficient interactive programming systems.” PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, 1970.
- [32] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: a transparent dynamic optimization system”. In: (2000).
- [33] John Aycock. “A brief history of just-in-time”. In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.
- [34] Dean Deaver, Rick Gorton, and Norm Rubin. “Wiggins/Redstone: An on-line program specializer”. In: *Proceedings of the IEEE Hot Chips XI Conference*. 1999.
- [35] Michael Gschwind et al. “Dynamic and transparent binary translation”. In: *Computer* 33.3 (2000), pp. 54–59.
- [36] Cindy Zheng and Carol Thompson. “PA-RISC to IA-64: Transparent execution, no recompilation”. In: *Computer* 33.3 (2000), pp. 47–52.
- [37] 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.
- [38] Mozilla Foundation. *SpiderMonkey*. URL: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>.

- [39] 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.
- [40] Adobe System Inc. *ActionScript 3.0 overview*. URL: https://www.adobe.com/devnet/actionscript/articles/actionscript3%5C_overview.html.
- [41] Ecma International. *Standard ECMA-262*. 1999. URL: <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.
- [42] Andreas Gal. “Efficient Bytecode Verification and Compilation in a Virtual Machine Dissertation”. PhD thesis. PhD thesis, University Of California, Irvine, 2006.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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.

- [47] Carl Friedrich Bolz and Laurence Tratt. “The impact of meta-tracing on VM design and implementation”. In: *Science of Computer Programming* (2013).
- [48] Carl Friedrich Bolz et al. “Meta-tracing makes a fast Racket”. In: *Workshop on Dynamic Languages and Applications*. 2014.
- [49] Carl Friedrich Bolz. “Meta-tracing just-in-time compilation for RPython”. PhD thesis. Heinrich Heine University Düsseldorf, 2012.
- [50] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. “Loop-aware optimizations in PyPy’s tracing JIT”. In: *ACM SIGPLAN Notices*. Vol. 48. 2. ACM. 2012, pp. 63–72.
- [51] Maarten Vandercammen. “The Essence of Meta-Tracing JIT Compilers”. In: (2015).
- [52] Spenser Bauman et al. “Pycket: a tracing JIT for a functional language”. In: *ACM SIGPLAN Notices*. Vol. 50. 9. ACM. 2015, pp. 22–34.
- [53] Michael Bebenita et al. “SPUR: a trace-based JIT compiler for CIL”. In: *ACM Sigplan Notices* 45.10 (2010), pp. 708–725.
- [54] Mike Pall. *DynASM*. 2012. URL: <https://lua-jit.org/dynasm.html>.
- [55] Peter Cawley. *The unofficial DynASM documentation*. URL: <https://corsix.github.io/dynasm-doc/>.
- [56] Mike Pall. *LuaJIT 2.0 intellectual property disclosure and research opportunities*. 2009. URL: <http://lua-users.org/lists/lua-l/2009-11/msg00089.html>.
- [57] Luke Gorrie. *Studio*. 2017. URL: <https://github.com/studio/studio/>.
- [58] OpenResty Inc. *OpenResty*. URL: <https://openresty.org/>.
- [59] Mike Pall. *LuaJIT Wiki*. URL: <http://wiki.lua-jit.org/Home>.

- [60] *Static Single Assignment Book*. URL: <http://ssabook.gforge.inria.fr/latest/book.pdf>.
- [61] Mike Pall. *LuaJIT 2.0 Bytecode Instructions*. URL: <http://wiki.luajit.org/Bytecode-2.0>.
- [62] Mike Pall. *LuaJIT 2.0 SSA IR*. URL: <http://wiki.luajit.org/SSA-IR-2.0>.