CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

TigerShrimp: An Understandable Tracing JIT Compiler

Master's thesis in Computer Science and Engineering

Jakob Erlandsson

Simon Kärrman

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

TigerShrimp: An Understandable Tracing JIT Compiler

Jakob Erlandsson
Simon Kärrman

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

TigerShrimp: An Understandable Tracing JIT Compiler

Jakob Erlandsson
Simon Kärrman

Supervisor: Magnus Myreen, CSE
Examiner: Wolfgang Ahrendt, CSE

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

TigerShrimp: An Understandable Tracing JIT Compiler

Jakob Erlandsson
Simon Kärrman
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Tracing Just-in-Time (JIT) compilers have been used for compiling programs that regular compilers struggle with. The difference is clearest in dynamically typed languages such as JavaScript and Python where very little is know about a program's behavior before run-time. During run-time, tracing JIT compilers look for loops in the code and only compile the most commonly taken path through these loops. Since compilation is a time consuming process, the idea is to only compile code that is run frequently enough for it to make up the time it takes to compile it. This works on the assumption that a program spends most of its execution time inside a few loops.

Commercial tracing JIT compilers are very complex systems integrating many functionalities into a highly efficient program. As such, the general ideas behind tracing JIT compilers can be quite difficult to understand from reading about these compilers. This project explores the possibility of creating a smaller scale tracing JIT compiler with the main objective of making it understandable. As far as we know, this is the first tracing JIT compiler with this stated objective.

Our tracing JIT compiler is designed for understandability meaning much effort was put into separating the different parts that make up a tracing JIT compiler into distinct modules, giving a clear overview of the responsibilities of the different parts and how they interact. Another aspect of understandability is the run-time visualizing toolkit developed alongside the tracing JIT compiler, this allows for insight into otherwise unseen functionality of the compiler.

Performance of the compiler is not the main objective of this project, despite this, we show results highlighting the potential of tracing JIT compilers where our compiler in some cases outperform the OpenJDK JVM.

Keywords: Compiler, JIT, tracing JIT compiler, programming languages, run-time visualization, designed for understandability.

Acknowledgements

We would like to thank Magnus Myreen for his support and feedback as supervisor for this project.

Jakob Erlandsson & Simon Kärrman, Gothenburg, June 2021

Contents

List of Figures

1 Introduction

Programming languages are used by programmers to get a computer to solve different tasks for them. However, computers cannot directly make sense of anything in a high-level programming language and as such, certain tools are required to bridge this gap between the programmers and the computer.

Bridging tools that can be used are, for example, interpreters and compilers. Interpreters are programs that interpret the code more or less as the programmer wrote it and evaluate it within the interpreter program. In contrast, compilers are programs that take the source code and translate it into code that can be run directly on the computer's processor. Such code is referred to as native code. For many programming languages, examples being C and C++, the entire source code is compiled before the program is actually run. This is referred to as Ahead-of-Time (AOT) compilation.

The code in Figure 1.1 illustrates an example where both interpretation and compilation have their benefits and shortcomings. Compiling this code will lead to the loop executing much faster than it would using interpretation; however, the two lines before the loop starts will also be compiled even though they only run once and thus there is not much benefit from compiling them. This would be more noticeable if the code before the loop was much longer. As such, one can see that this code could benefit from first interpreting the code that is only executed once and then only compiling the loop.

```
void foo() {
  int i = 0;
  int j = 0;
  while (i + j < 100000) {
    i += 1;
    j += 2;
  }
}
```

Figure 1.1: Small example program that in different ways benefits from interpretation and compilation.

Just-in-Time (JIT) compilers differ from conventional AOT compilers in that they do the compilation while running the program rather than before. In addition to this, they only compile the functions or blocks of the code which are the most performance critical, meaning the ones that are executed the most. These compilers have gained popularity lately, in part due to their inclusion into most of the main-stream web engines [2, 3]. Their usefulness in web applications lies in their ability to start interpreting immediately and optimizing parts of the code over time through compilation, leading to fast start up times and eventually optimized programs. Additionally, they can take advantage of the dynamic properties of languages which are determined at run-time. This is something AOT compilers cannot do since they can only make use of a program's static information, which for dynamic languages is not much.

Tracing JIT compilers have all the same benefits as a regular JIT compiler but base their optimizations on *traces* rather than blocks or functions. A trace is a sequence of statements taking exactly one path through the code. Shown in Figure 1.2 is an example where the red dotted line indicates a single trace through a block of code. The motivation behind this approach follows from two general observations [4]:

1. A program spends most of the time in just a few *hot loops* in the code.
2. Within these loops, where the code can branch to take several paths through the loops, one of these paths, or traces, is much more commonly taken than the other ones.

Note that a trace consists of all instructions starting at a loop header and ending at the end of the loop. Therefore, a trace can span multiple functions as a loop can contain function calls.

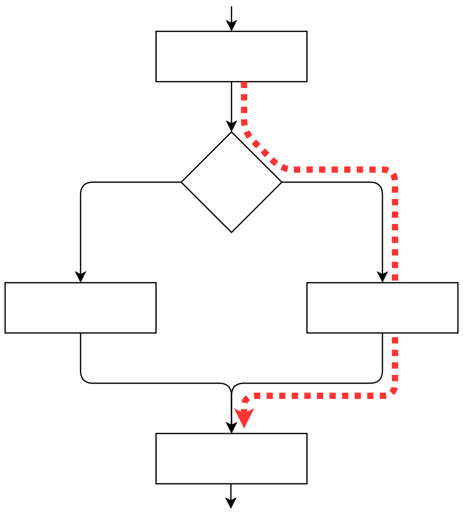


Figure 1.2: A single trace through a code sequence.

Where an AOT compiler only needs to produce native code from source code, which in itself is a rather complex procedure, a tracing JIT compiler additionally needs an interpreter for source code, a way to execute native code and a way to switch between interpreting, compiling and executing native code. One complexity introduced by this approach is that an interpreter for source code and an interpreter for native code (also known as a Central Processing Unit or CPU) operate in very different settings and abstraction levels. Despite this they need to have a way to communicate information between each other to make this mixed evaluation method function. This makes tracing JIT compilers more complicated to understand and maintain.

This thesis reports on our project which set out to explore the possibility of creating an easily understandable tracing JIT compiler by showing how the implementation can be written to maximize understandability and by visualizing the run-time behavior of the compiler. We define some criteria for understandability:

- A clear definition of the responsibilities of the different parts that together form a tracing JIT compiler and how they interact with each other.
- A high degree of separation of concern between these different parts.
- A strong resemblance between the theoretical description, and our implementation, of a tracing JIT compiler.

At the time of this writing, we have not been able to find any evidence of an attempt to create a tracing JIT compiler with this main objective.

1.1 Contributions

This thesis shows that it is feasible to develop a tracing JIT compiler designed for understandability, focusing on a clean and simple implementation. We have also developed a run-time visualization tool that shows the behavior of our compiler while evaluating a program which exposes some otherwise hidden functionality from the viewer. We highlight understandability both in implementation and in extracting run-time information to more clearly show how the compiler switches from interpreting code to compiling and directly executing native code.

1.2 Limitations

The work presented in this thesis project is subject to the following limitations:

- To reduce the scope and focus to mainly the tracing JIT compiler, we consider a byte-code language as input rather than a high-level programming language.
- While tracing JIT compilation of a dynamic language would be interesting as this is where most of the commercial tracing JIT compilers operate, we opted to target Java because its byte-code has clear and well documented semantics.
- We only support a subset of the functionality of Java where only public static methods written in a single file are included. This limitation makes the thesis more focused on developing the tracing JIT compiler rather than on the specific language it implements.
- Similarly, we do not support traces containing function calls.
- When compiling traces, we will only consider generating native machine code targeting the x86-64 architecture.

2
Background

Implementation of a tracing JIT compiler requires familiarity with a number of concepts:

- Implementing a programming language: this can involve interpretation (2.1) and/or compilation of the source code as well as combining the two (2.4);
- Machine languages and virtual machines: for this project, this involves the Java Virtual Machine (JVM) (2.5) and x86-64 (2.3);
- Profiling (2.2) and debugging (2.6): methods for extracting and visualizing run-time behavior.

This chapter introduces these concepts.

2.1 Interpreting

An interpreter is a program that directly evaluates source code without first compiling it into native code. Interpreting can be done during several stages of code transformation and there are more than one approach for interpreting at any given stage. In this section we go through some of these approaches.

2.1.1 Abstract Syntax Tree Interpreter

An early step in the evaluation of source code is to parse the code and construct an abstract syntax tree (AST) which is a grammatical representation of the source code with no information about how to execute it [5]. This structured syntax can then directly be interpreted. Transformation into AST is done regardless of whether the code is to be compiled in any form or interpreted.

This approach leads to very fast startup times since we only have to parse the code to see that it is syntactically correct before the interpreter starts evaluating. Furthermore AST interpreters are easy to understand and simple to build and maintain. Because of this, they are often used as a first step when implementing an interpreter or compiler for a language [6]. They do have their drawbacks though. Since they do not contain any information about how to execute the source code, they have to do a lot of unnecessary work on nodes that do not contain any useful information. Take this example from the SquirrelFish announcement:

> ”For example, for the block `{ x++; }`, the interpreter would first visit the block node `{...}`, which did nothing, and then visit its first child, the increment node `x++`, which incremented `x`.” [5]

This means that for any practical purpose, one would want to continue developing their interpreter past this point for increased effectiveness. Compiling the source code into an intermediate byte-code language gets rid of many of these non-functional operations and leaves the interpreter to only evaluate what actually matters to the final result.

2.1.2 Pure Byte-Code Interpreter

A byte-code language is a lower level representation of a high-level language but still not machine readable. This language can be directly interpreted or further compiled. A byte-code interpreter relies on that an AST is converted into a sequence of byte-codes where every byte-code has a clear and predefined function. A byte-code instruction consists of a few bytes where the first byte has a corresponding mnemonic describing the functionality of this instruction. The following bytes are parameters to the instruction. With this the main functionality of a byte-code interpreter is very simple. Each byte-code is implemented in an addressable form, e.g, a switch/case structure. The interpreter looks at the byte-code at the current position of the program counter and moves the control to the code implemented for that byte-code. This is done in a loop that loops forever so every byte-code implementation need to return control to the interpreter loop at the end of its implementation [7].

Every byte-code is, as the name might suggest, represented in memory by exactly one byte and thus it can take 256 values. Even though 256 different operations is quite many, it still is a limitation on how specific one can make every operation. Because of this, the operations defined in a byte-code language needs to be quite general.

2.1.3 Threaded Code Interpreter

Where as a byte-code interpreter constantly needs to read from memory which byte-code instruction is to be executed next, a threaded code interpreter instead jumps directly to the next implementation, saving a memory lookup for the address of the next one, until the program terminates. This saves some overhead at every byte-code instruction and most importantly skips one memory access instruction and one jump instruction which are both fairly expensive [7].

For further optimization, one can analyze the byte-code sequence to see if the same sub-sequence of codes are often executed after each other. In this case one can extract the resulting functionality of executing these after each other and simply execute the result instead, saving on even more overhead [7].

The main goal of a threaded code interpreter is to be fast. With this it instead lacks some understandability and simplicity when it comes to implementation compared to a pure byte-code interpreter.

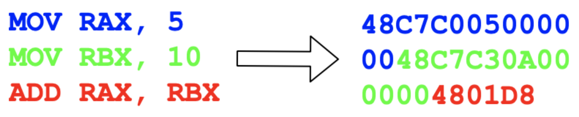
2.2 Profiling of Hot Traces

There are techniques for finding (with a high degree of confidence) the actual most common trace through a loop [8, 9] and there are applications that, during run-time, use these techniques to increase performance [10]. However, for tracing JIT compilers it is usually the case that the process of finding these correct traces is more costly than the benefit of finding them. Instead an assumption is often made that the first path taken is a common one and thus only this is compiled at the start [11, 4, 12].

Since multiple paths through the loop can exist, we still need to handle the cases where the evaluation does not follow the path that was compiled. These cases will be referred to as *side exits* from the trace.

2.3 Native Code Assembly

On the left hand side of Figure 2.1 is what we usually refer to as "machine code". However even this is an abstraction made to make it more readable. Code that a processor can read directly looks more like on the right hand side of Figure 2.1, which is simply a string of bytes, here in hexadecimal form for compactness. The process of converting machine code to its corresponding bytes is what we call assembling.



```
MOV RAX, 5          48C7C0050000
MOV RBX, 10         0048C7C30A00
ADD RAX, RBX        00004801D8
```

Figure 2.1: Conversion of x86-64 assembler into bits, represented here as hexadecimal codes.

2.4 Dynamic Native Code Execution

Dynamic native code execution refers to running a piece of native code that has been generated and saved in memory rather than on disc, meaning we can execute it from within the run-time environment of a program.

The main issue here is to make sure that the state of the interpreter gets transferred correctly, both before and after a trace has been executed. The state consists of the program counter, the variables in the local store and the values on the stack. All of these need to be placed so that the native code can make use of them and needs to be returned correctly when the trace ends, either normally or via a side exit.

One approach to keeping the interpreter state synchronized while running native code is to use an external data structure to communicate the values and state between the interpreter and the native code. The reference to this data structure can then be reached from within the native trace and thus all of its values can as well. The trace updates the values in the external data structure so it is up to the interpreter to update the state according to these values [12]. Another approach is by letting the native trace have direct access to the interpreter state by passing a reference to it, leaving the native code responsible of updating the state before exiting [13].

Regardless of which approach is used to updating the values in the interpreter state, some more information needs to be communicated from running the trace. We must identify which exit was used to make the interpreter start at the correct location in the code. Furthermore, when a side exit is used, we need to check if this exit point has become hot and start recording if it has.

2.5 Java Virtual Machine

The Java Virtual Machine (JVM) is an representation of a hardware computer in software. It has an executable instruction set and can manipulate a stack and a variable store which keeps track of values needed to perform the necessary calculations. Several languages, but most notably Java, are compiled into this intermediate representation before either being compiled further into native machine-code or interpreted by a specialized program. The byte-code instructions are a part of the class file produced by a Java compiler. Apart from the actual code, this class file contains a constant pool with information about methods, variables and constants as well as other, at times, useful information.

2.5.1 Operand Stack & Variable Store

The JVM is a stack based virtual machine. This means that it does not have any virtual registers or similar where run-time values can be directly modified. Instead all operations are done on values located on the operand stack. The operand stack is a last in first out stack and is specific for every instance of an evaluated method. This means that the only operations that can be done on the stack is essentially push and pop so the only directly accessible value on stack is the one that was pushed last. For example, if one wants to evaluate the expression $2+5$ in the JVM, 2 is first pushed onto the stack followed by 5. Then the addition operation will take the two topmost operands on the stack and replace them with the result of the operation.

To keep track of the run-time values of a method's local variables a variable store is used and JVM provides operations to move values from the top of the stack to the variable store and vice versa. The local variable store has an array-like structure where values are accessed via indexing. All the entries in the variable store occupies 32 bits so in order to store 64 bit value like a Long or Double, one has to use two entries in the store. This means that if a 64 bit value is placed at index i is the next available entry at index $i+2$.

The stack is empty at the beginning of evaluating a method and the arguments to the method are placed in the variable store starting at index 0. When calling a method, the arguments are placed onto the callers stack. The JVM then moves these values from the callers stack into the variable store of the callee method. When the method returns, the arguments are replaced by the return value of the called method.

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 2.2: Structure of the Java class file.

2.5.2 The Java Class File

The class file is a series of bytes, Figure 2.2 describes the basic structure of it. Here u denotes how many bytes needs to be read to retrieve the given information. The entries cp_info, field_info, method_info and attribute_info have their own separate structures that are read in a similar way. For our purposes, we only need to read the constant pool, the methods and some of the attributes in order to create a functioning interpreter. The actual byte-codes are located in one of the attributes called code_attribute which in turn is part of the method_info.

2.5.2.1 Constant Pool

Each entry in the constant pool has a tag describing what kind of entry it is as well as some extra bytes. The number of extra bytes and what they contain differs depending on the tag.

2.5.2.2 Method Info

The access flag, referenced in Figure 2.3, describes if the method is public, private, static, etc. Name index and descriptor index refers to entries in the constant pool. The name index gives us the name of the method and the descriptor index gives us the return type as well as the types of the method's arguments. The method descriptor string is of the form $(T_1T_2)T_3$ where T_1 and T_2 are argument types and T_3 is the return type. Each method must have exactly one return type and zero or more argument types. To extract the actual byte-codes that represent the functionality of the method one needs to parse the attribute_info entry and more precisely the

```
method_info {
   u2             access_flags;
   u2             name_index;
   u2             descriptor_index;
   u2             attributes_count;
   attribute_info attributes[attributes_count];
}
```

Figure 2.3: Structure of the method info entry in the Java class file.

code_attribute which is described in 2.5.2.3.

2.5.2.3 Attribute Info

An attribute entry is similar to a constant pool entry in that it has a tag describing what type of attribute it is. There are many different attributes that are more or less important depending on how much of the Java language one wants to support. For example, there are entries only used for debugging purposes and others for handling imported modules. The only one we need in order to parse the program is the code attribute.

```
Code_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 max_stack;
   u2 max_locals;
   u4 code_length;
   u1 code[code_length];
   u2 exception_table_length;
   {   u2 start_pc;
       u2 end_pc;
       u2 handler_pc;
       u2 catch_type;
   } exception_table[exception_table_length];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Figure 2.4: Structure of the code_attribute.

From the structure, shown in Figure 2.4, we extract some information. The `code` list, which format is described in more detail in 2.5.3, as well as the `max_stack` and `max_locals` values.

2.5.3 Java Byte-Code

Here, one byte represents either an instruction mnemonic or a parameter to an instruction. Each instruction has a specific number or parameters, usually zero, one or two but some special instructions have many more. The functionality of these instructions can be divided into different categories such as constant, load and arithmetic instructions. They have more or less descriptive names and further descriptions of their intended functionality can be found in documentation [14]. Most of the instructions operate only on the stack, variable store and/or the program counter and are as such, very easy and straight forward to implement. Where it gets a little more tricky is when we need to handle references, method calls as well as some long and double valued constants. In this case we also need to access some locations in the constant pool.

2.6 Binary Debuggers

A debugger is a program, like the name might suggest, that is often used to eliminate bugs in programs. The way a debugger does this is to show the developer the run-time behavior of their program and in that way help to understand why a bug occurs. Debuggers support a wide range of operations, among these are the three most basic operations: Starting a program within the debugger, pausing the program execution when predefined conditions are met, and showing the program's state. The state in this sense can refer to, for example, types and values of variables defined within the program and also underlying state of the system such as what values are located in actual registers of the computer and the contents of the stack and memory [15].

The way a debugger can show the state of variables in a program is not a straight forward task. This is because a debugger is most often provided a program executable in binary form where most of the context of variables have been removed by the compiler during compilation. So the task of the debugger is to reconstruct the original program with variables and statements from its internal state. To make this possible the compiler will, if asked, generate debug information during compilation. This information is structured in a way which makes it possible for the debugger to reconstruct the original program at run-time. This information contains, for example, names and types of variables as well as in which module and file it was first defined. It also contains information about methods and also decisions made by the compiler, for example, how it decided to store a structure in memory [15].

There are three major C++ debuggers targeting different platforms. Microsoft's Visual Studio debugger mainly targets Windows [16], GNU's GDB which is the de facto standard for Unix systems [17] and LLBD which comes pre-installed on MacOS systems [18].

3

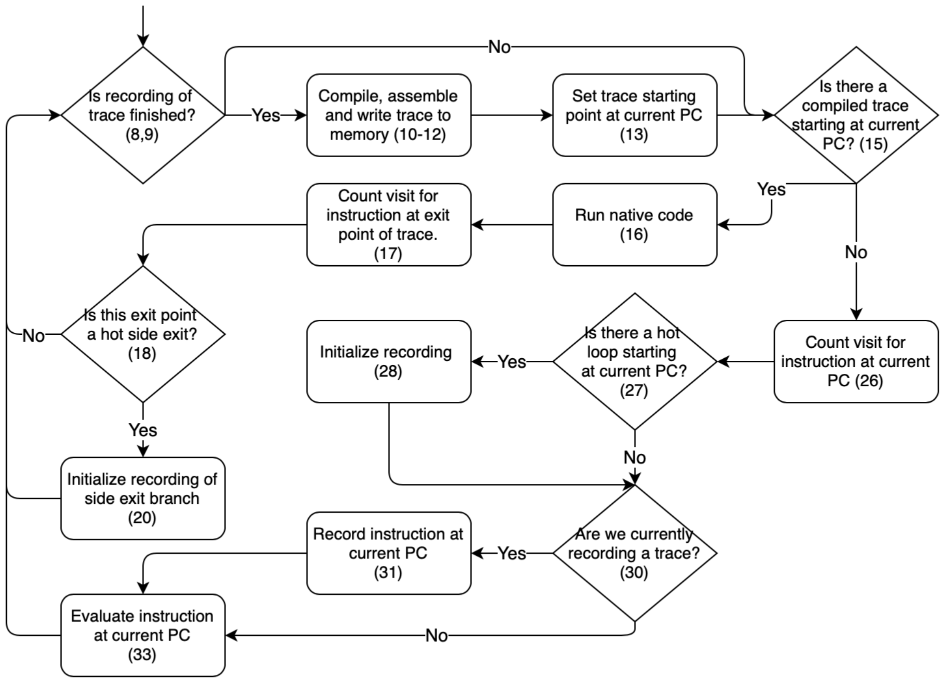
Developing a Tracing JIT Compiler for JVM

This chapter will provide a high-level description of a tracing JIT compiler as well as an in-depth explanation of the different steps required to implement a tracing JIT compiler.

3.1 A Tracing JIT Compiler

A tracing JIT compiler consists of several parts handling the different stages of evaluation. The different stages are interpretation, profiling of hot loops, recording of traces, compilation of traces and execution of native code. In our tracing JIT compiler, there is a central run-time loop that dispatches the modules handling these stages.

Figure 3.2 shows the central run-time loop in the code and Figure 3.1 shows the general control flow of it.



Is recording of trace finished? (8,9)
Yes
No
Compile, assemble and write trace to memory (10-12)
Set trace starting point at current PC (13)
Is there a compiled trace starting at current PC? (15)
Yes
Run native code (16)
Count visit for instruction at exit point of trace. (17)
No
Is this exit point a hot side exit? (18)
No
Yes
Initialize recording of side exit branch (20)
Count visit for instruction at current PC (26)
Is there a hot loop starting at current PC? (27)
Yes
Initialize recording (28)
No
Are we currently recording a trace? (30)
Yes
Record instruction at current PC (31)
No
Evaluate instruction at current PC (33)

Figure 3.1: Control flow of run-time loop. Numbers in parenthesis references line number in Figure 3.2. Note: PC stands for program counter.

A tracing JIT compiler starts by interpreting a sequence of byte-code instructions. During interpretation, it keeps track of every time the program enters each loop in the interpreted byte-code instructions and increments a counter associated to that specific loop. Once a loop has been entered a predetermined number of times, it is considered hot and recording starts.

Recording produces a trace that describes exactly one path through the code but it is not necessarily the case that this path will be taken every time the code is run. To this end, we must ensure that we can interrupt the execution of a trace if an alternate path is taken and give back control to the interpreter which starts up again at the point of exit.

During recording, each instruction and the state of the interpreter is saved at every iteration of the run-time loop. This is done for every instruction until the program counter returns to the loop header at the start of the trace. These instructions can then be compiled and saved in run-time memory so that the next time the program counter reaches the loop header, the interpreter gives up control and the compiled code will run instead.

In the event of an early exit, we treat the next instruction as a loop header regardless of what it actually is. This is because if a trace takes an alternate path a high number of times, then we have several hot traces through one loop and we will benefit from compiling all of them. We must construct these traces so that when a side exit is taken into a different trace, we do not give back control to the interpreter before running the next trace but handle this inside the trace itself. This saves time due to reduced overhead.

```cpp
void RunTime::run(Program *program) {
  initProgramState(program);
  // Having an empty state stack means the evaluated program has
  // reached the return statement in its main method and thus has
  // finished and we can terminate.
  while (!program->states.empty()) {
    State *state = program->states.top();
    ProgramCounter pc = state->pc;
    if (traceRecorder.isRecording() &&
          traceRecorder.recordingDone(pc)) {
      Recording recording = traceRecorder.getRecording();
      Trace trace = compiler.compileAndInstall(
          program->methods[pc.methodIndex].maxLocals, recording);
      traceHandler.insertTrace(recording.startPc, trace);
    }
    if (traceHandler.hasTrace(pc)) {
      ProgramCounter exitPc = traceHandler.runTrace(state);
      profiler.countSideExitFor(exitPc);
      if (profiler.isHot(exitPc)) {
        ProgramCounter loopHeaderPc = pc;
        traceRecorder.initSideExitRecording(loopHeaderPc, exitPc);
      }
      state->pc = exitPc;
    }
    else {
      ByteCodeInstruction inst = interpreter.prepareNext(program);
      profiler.countVisitFor(pc);
      if (profiler.isHot(pc)) {
        traceRecorder.initRecording(pc);
      }
      if (traceRecorder.isRecording()) {
        traceRecorder.record(pc, inst);
      }
      interpreter.evalInstruction(program, inst);
    }
  }
}
```

Figure 3.2: Run time loop written in C++ code.

3.2 Interpreter

In this section we will go through the steps we took to constructing an interpreter capable of evaluating JVM byte-code produced by a Java compiler.

3.2.1 Java Class File Parser

Most of the fields in the Java class file are unnecessary for our purposes, either way we must parse them to keep track of where in the file we are, but we do not need to save them. Some fields require a bit more analysis than others.

Constant Pool

The entries in the constant pool represent different parts of the source code but in the class file they are all structured the same way. As such, we read the entries as specified and save the information in a super class called CPInfo. We save these in this more general form since the order of these entries are important to preserve. This because when entries in the constant pool are referenced in the actual code it is only by the index of the entry.

When we need to access something from the constant pool during interpretation, we will know from the context which type of CPInfo is supposed to referenced to. Exceptions to this is the CONSTANT_Long_info and CONSTANT_Double_info entries as the byte-codes instruction does not give any hint as to which one to use and their data is extracted the same way. This means in order to figure out what type the value is supposed to have, we must look at the corresponding tag of the entry to figure it out. The same holds for CONSTANT_Integer_info and CONSTANT_Float_info.

One thing to note about the constant pool is that every constant except for two special ones takes up exactly one entry in the constant pool, even if some of them are lists or string constants. These two are the long and double constants that take up two entries which is something to pay special attention to. As Lindholm et. al. put it:

> ”In retrospect, making 8-byte constants take two constant pool entries was a poor choice.” [14]

Another odd thing about the constant pool is that the first entry is at index number 1 which is unlike most programming conventions and thus can lead to confusion if not considered.

Attributes

The JVM specification states that in order to correctly interpret a JVM program there are six special attributes needed to be able to handle. However, for the purpose of this thesis project the only attribute needed is the Code_attribute, which is shown in Figure 2.4.

Program structure

We transform the contents of the parsed class file into the form in which we define a program. Figure 3.3 shows two data structures, *Program* and *Method*, they contain all the information that is needed in order to correctly evaluate the input program. *Type* is an enumeration that represents the types that are currently supported, which is Int, Float, Long and Double. The `states` stack represents the run-time state of the interpreter. The top item on this stack contains the program counter, operand stack and variable store for the method currently being interpreted. When a method is called in the program being interpreted, a new state is placed on top of the `states` stack, when a method returns, the state on top of the `states` stack is reomved.

```
struct Method {
  int nameIndex;
  Type retType;
  std::vector<Type> argTypes;
  int maxStack;
  int maxLocals;
  std::vector<uint8_t> code;
};

struct Program {
  std::map<int, CPInfo*> constantPool;
  std::map<int, Method> methods;
  std::stack<State*> states;
};
```

Figure 3.3: Structure of our parsed program.

3.2.2 Byte-Code Interpreter

Our interpreter is implemented in such a way that it takes a state as a parameter and modifies its stack and variable store according to what the next instruction in the state is. Rather than allowing the interpreter itself to be the run-time loop, we instead let the main run-time loop utilize the interpreter when compiled traces are not available. This makes us more able to provide a clear description of the functionality of our program, which is shown in Figure 3.2.

The operand stack and variable store can contain variables of any number of different types at the same time and in no particular order. As such, we need a way to represent any value with a single type. We created the *Value* struct seen in Figure 3.4 so that every type does not require its own stack and variable store. The *union* keyword means that the struct can assume exactly one of the types described and will always reserve enough memory for the largest type. This practice of encapsulating values in more general types is commonly referred to as boxing [13].

```
struct Value {
  Type type;
  union Data {
    int intValue;
    long longValue;
    float floatValue;
    double doubleValue;
  } val;
};
```

Figure 3.4: Structure of the interpreter's internal representation of a value.

Because of this abstraction many of the type-specific byte-code instructions can be done in the exact same way using this struct, Figure 3.5 is an example of this where ”ILOAD_0” is the instruction that takes the int value from address 0 in the variable store and puts it on top of the operand stack,"FLOAD_0" is the same but for a Float value and so on.

```
case ILOAD_0:
case LLOAD_0:
case FLOAD_0:
case DLOAD_0: {
    program->load(0);
    break;
}
```

Figure 3.5: Example of a byte-code instruction which does the exact same thing for many types.

Expressions are evaluated left to right and as such we need to pay attention to the order in which we push and pop values to the stack. Take as an example the expression `6/3`, the equivalent Java byte code is:

```
ICONST 6
ICONST 3
IDIV
```

When the `IDIV` instruction is evaluated, the stack contains a `3` and a `6`, in that order. We can only access the topmost element of a stack so the first value popped will represent the right hand side of the division expression. This order of evaluation is the same for all binary operations, but as some are reflexive, for example the addition function, it does not always matter.

3.3 Profiler

In order to find hot loop headers, we assume that every branch instruction that moves the interpreter's position in the code backwards has a loop header as its target. The interpreter's position in the code is described by both the index of the method currently being interpreted and the program counter within that method. Therefore we need a method index in addition to the program counter to distinguish the program counter's position in the code.

The program counter at the first instruction in each method is zero. Because of this, a method call would look like a branch back to the start of a loop header if we only looked at the program counter, therefore only branches backwards which do not affect the method index are counted. The profiler keeps track of the number of jumps to each loop header and once this number is higher than a predetermined threshold (in our case 2) that loop is considered hot and will be recorded.

3.4 Recorder

In essence, recording of a trace involves saving each byte-code instruction and the program counter at each instruction until the start of the trace is reached again.

However there are some additional things we can do at this stage to help simplify the compilation later on.

During recording, we pay special attention to jump instruction. The parameter to jump instructions is relative to the position of the instruction, as such it is negative for backward jumps and positive for forward jumps. A trace takes exactly one path through the code and conditional jumps move the program counter to two possible locations: either directly after the jump instruction, we call this the fall-through option, or to some instruction further ahead. One of these jumps is always out of the trace and to simplify compilation we want the fall-through option to always continue in the trace. This means that we might have to invert the direction of some conditional jump instructions when recording.

On the other hand, when it comes to unconditional jump instructions, we interpret backward jumps as jumps to the start of the trace while unconditional forward jumps within a trace have no semantic meaning as we can simply read the instruction it jumps to as the next recorded instruction. Because of this, we omit these forward branch instructions during recording and compilation.

We need a way to distinguish between the different jump targets, both to keep track of different side exits and in order to correctly place labels in the compiled code. As such, we calculate the target of each jump instruction depending on its offset. Together with the recorded trace, we provide the compiler with a list of these jump targets.

Furthermore, some of the byte-code instructions, for example, ISTORE which usually takes a parameter, also have some specific parameter-less variants for the lowest addresses in order to reduce the size of the class file. These are for example ISTORE_0 and ISTORE_1. While recording, we convert all the parameter-less versions of these byte-code instructions into parameterized ones to reduce code duplication in our compiler's source code.

Lastly, in the event that we start recording instructions that are outside of the loop, we must immediately stop recording as we might never return to the loop header. This can happen if the recording starts on the exact same iteration on which the looping would end. This is solved by restarting the recording if a new hot loop is encountered before the first recording finishes.

3.5 Compiler

Our compiler takes a series of byte-code instructions and translates them into a series of native x86-64 instructions. Our representation of a native instruction, shown in Figure 3.6, consists of a mnemonic and two operands. For some mnemonics, one or both of these operands are not set as that mnemonic has zero or one operands. Our operands are similar to the *Value* structure, described in Section 3.2.2, in that they consist of a type and a field of data. These types are register (general purpose and xmm), memory location, immediate value or label.

The recorded byte-code instructions are compiled in a single pass from top to bottom. Since what we are compiling is a stack-based byte-code language, the compiler needs an internal stack in order to keep track of the values used in certain instructions. As a result of this, we do not generate native code for every byte-code

```
struct Instruction {
  x86::Mnemonic inst;
  Op op1;
  Op op2;
};
```

Figure 3.6: Internal representation of a native instruction.

instruction by itself. For example, the ICONST instruction simply places a value on top of the compiler's stack to be used at a later point, where as an instruction like ILOAD actually generates a MOV instruction to place a value in a usable register.

The recorded trace contains a number of branch instructions. All of these, except the one that branches back to the start of the trace, are branches that will end the execution of this trace. We call these exit points, how we handle exiting a trace is described in detail in Sections 3.8 and 3.9. The compiler needs to generate code for each of these exit points to communicate to the rest of the program which exit was used.

3.6 Assembler

The task of the assembler is to take the native instructions produced by the compiler and create a corresponding string of bytes to send to the processor. For the purpose of efficiency, modern processor architectures can have different byte representations corresponding to the same mnemonic depending on what its operands are. For example, in the two instructions `MOV RAX, RBX` and `MOV RAX, 10`, the `MOV` mnemonic will be encoded to `0x89` and `0xc7` respectively. Because of this, the task of building an assembler from scratch is beyond the scope of a project this size so we opted to use the C++ library called "AsmJIT" instead [19]. AsmJIT provides us with tools to create the byte string corresponding to the native instructions that we give it while still being flexible enough to let us handle memory allocation and native execution ourselves.

3.7 Executing an Assembled Trace

Provided a list of bytes there are some steps to be taken in order to be able to execute it as native code.

We check whether we need to allocate more memory to store the trace. Whenever we need to allocate more memory we can only do it in contiguous chunks of memory of predetermined size, so called pages. Therefore we need to look at how many bytes the trace consists of in order to know how many pages are needed to accommodate the entire trace. We also need to make sure that these memory pages are both writable and executable which is simple enough in C++.

Since we need to allocate whole pages and a page is large relative to the size of most traces (on most modern architectures, a page is 4096 bytes) it is likely that we can fit more than one trace onto a single page. We accomplish this memory handling by using a cursor to keep track of the last written position in memory and compare it to where the page ends in order to figure out if the trace will fit on the page.

Once the memory allocation is figured out, we can start writing our trace into sequential memory. We store the memory address where the trace starts into the `startAddr` pointer seen in Figure 3.7. After the trace is written, we can use the function pointer `execute` to call the trace as normal C++ function in which case the native code will be executed until exiting the trace.

```
typedef long (*pfunc)(void*, void*);

union TracePointer {
  pfunc execute;
  uint8_t* startAddr;
};
```

Figure 3.7: Data structure making it possible to call an address in memory as a regular C function.

3.8 Mixed Execution

Evaluating a program with a tracing JIT compiler requires us to interpret bytecode instructions as well as executing native code and switching between these two modes. Since we are evaluating the same program with both these methods, we need to make the variables in the variable store accessible and modifiable from within the trace.

Making the variables in the variable store accessible to the native trace involves creating a list of the values and passing this list as an argument to the trace. The index of each value in this list is its corresponding key in the variable store. Since we know the types of all variables during compile time, we can create the appropriate native instructions to correctly interpret the values of the variables. From within the trace we can now access each value as they are all placed sequentially in memory before the trace is called.

In addition to the list of local variable values, a native trace also needs access all other compiled traces available, the reason for this is explained in Section 3.9. All the external information needed by the trace is contained in the data structure in Figure 3.8, which is passed as an argument to the native trace.

```
struct ExitInformation {
  Value::Data* variables;
  uint8_t** traces;
};
```

Figure 3.8: Information needed from within the trace that gets passed as an argument when a trace is called.

When exiting the trace, the values that we read before entering the trace are written back to the variable store at their correct location which we get from their index in the list. In addition to this, we also need to change the program counter so that the interpreter starts reading from the correct location in the byte-code instruction sequence. Since a trace can have multiple exit points, we assign each exit point an id, this id is the return value of the trace. We can then use this id to look up where the interpreter is supposed to continue by referencing a map constructed during compilation.

3.9 Trace Stitching

Different parts of the code get compiled during evaluation, if one of these parts starts where another one ends, we can execute both of them as if they were one continuous trace. This saves some time as switching between the interpreter and executing native code comes with quite a lot of overhead. One situation where a trace stars where another one ends is when a side exit from a trace is taken enough times to be considered hot.

When a side exit gets hot, it will be recorded and compiled like any other trace. Since we only compile the side exit and do not recompile the original trace, the two traces are now separate and as such we need a way to run the side exit trace without having the interpreter take back control in between.

If there is a compiled trace starting at the interpreters current position, the interpreter will call this trace as a function. When a trace reaches an exit, the trace will call the helper function `handleTraceExit` seen in Figure 3.11 providing a unique exit ID depending on which exit was reached. If there is a compiled trace at this exit ID, this trace will be called by the helper function. If not, the exit ID will be returned to the interpreter.

The `handleTraceExit` function is called each time any trace reaches any exit, as such, the chain of function calls originating from the interpreter can become very long. The return value is that of the last called function and this value is not affected by the chain of function calls and as such it is very inefficient, both in terms of memory and time consumption, to carry this return value back through the chain of calls. Figure 3.9 illustrates this chain.

To reduce the computational overhead produced by calling this many functions, we apply tail calls to and from the `handleTraceExit` function. To this end, the function in Figure 3.11 is never actually called but instead the control flow is explicitly moved to the start of this function each time a trace reaches an exit. Similarly, this function doesn't call traces as functions but instead jumps to the address where the trace starts. Using this approach, only the last function in the chain returns and since the only function call made was from the interpreter, the return address is in the interpreter as expected. This approach is illustrated in Figure 3.10.

During trace compilation, it is easy to replace a call instruction by a jump instruction to achieve the behavior shown in Figure 3.10. Most C++ compilers are able to perform this optimization but to ensure that the `handleTraceExit` function always uses tail-calls, we decided to write an equivalent function to the one shown in 3.11 in inline assembly.

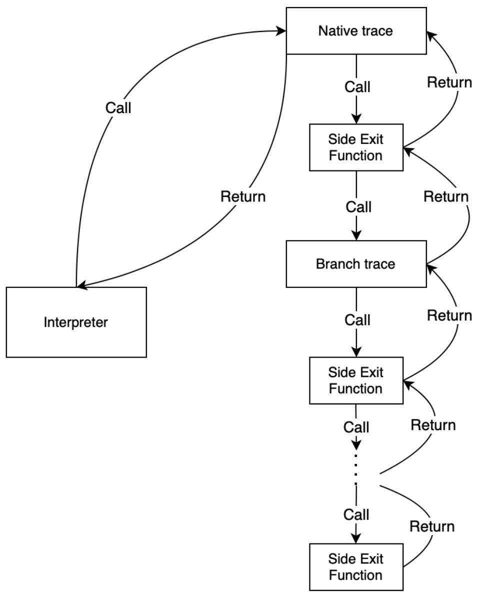


Figure 3.9: Trace stitching using function calls.

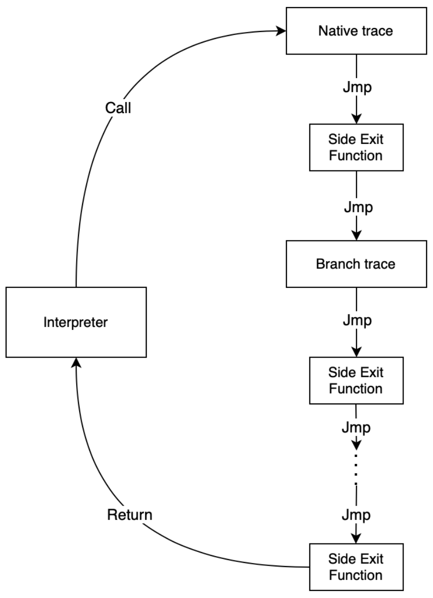


Figure 3.10: Tail-call optimized trace stitching.

```
int handleTraceExit(ExitInformation* info, int exitId) {
  if (info->traces[exitId] != 0) {
    TracePointer tracePointer;
    tracePointer.startAddr = info->traces[exitId];
    return tracePointer.execute((void*)info,
                                (void*)(&handleTraceExit));
  } else {
    return exitId;
  }
}
```

Figure 3.11: Trace exit handling function.

4
Visualizing Run-Time Behavior

To illustrate the workings of the tracing just-in-time compiler, we have developed a visualization toolkit that makes it possible to see what the tracing JIT compiler is doing. We developed the visualizer to be separate from the compiler, both to keep the compiler as simple as possible, and for the compiler to work entirely regardless of whether or not the visualizer is running. As such, we designed the visualizer as a program which in turn runs the compiler on some source code.

4.1 Implementation

An overview of how the visualizer operates is shown in Figure 4.1. The visualizer takes some source program as input and runs the compiler on that source while extracting information from the compiler's run-time. We achieve this by using a binary debugger for C++, specifically LLDB, which allows us to set break points at certain points of the run-time loop seen in Figure 3.2. We can expose the values of variables in all different components as well as the processor's registers. The raw output of LLDB, while structured and informative, is not very readable, see Figure A.1.

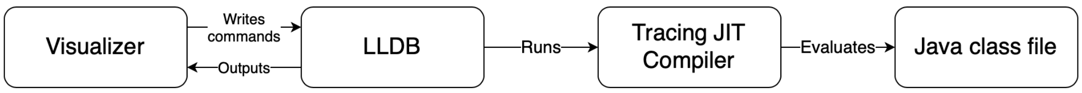


Figure 4.1: Chain of operations from the visualizer program.

We use unix pipes to make our visualizer program write to standard input of the LLDB process and to redirect LLDB's standard output back to the visualizer. We want to know the values exposed by LLDB in every iteration of the run-time loop, LLDB provides functionality for this in the form of stop hooks in which we can define some commands that should be run every time LLDB stops at a break point.

We chose to implement the visualizer in Python since most of the work it is doing is handling strings which is very easily done in Python. Performance is not a big factor in the visualizer as the debugging process is comparatively slow, so the other benefits of Python make up for the lower performance in this case.

4.2 Example

In this section, we will show the behavior of our tracing JIT compiler using our visualizer. The method that is evaluated is shown in Figure 4.2 and Figure 4.3 shows the byte-code instructions generated by compiling it. This method is interesting as it illustrates the functionality of the different parts of our tracing JIT compiler. There are three paths through the loop and they will all, at some point, be considered hot. This means that there will be three separate phases of recording and compiling during execution. An example in video form can be found on YouTube[1].

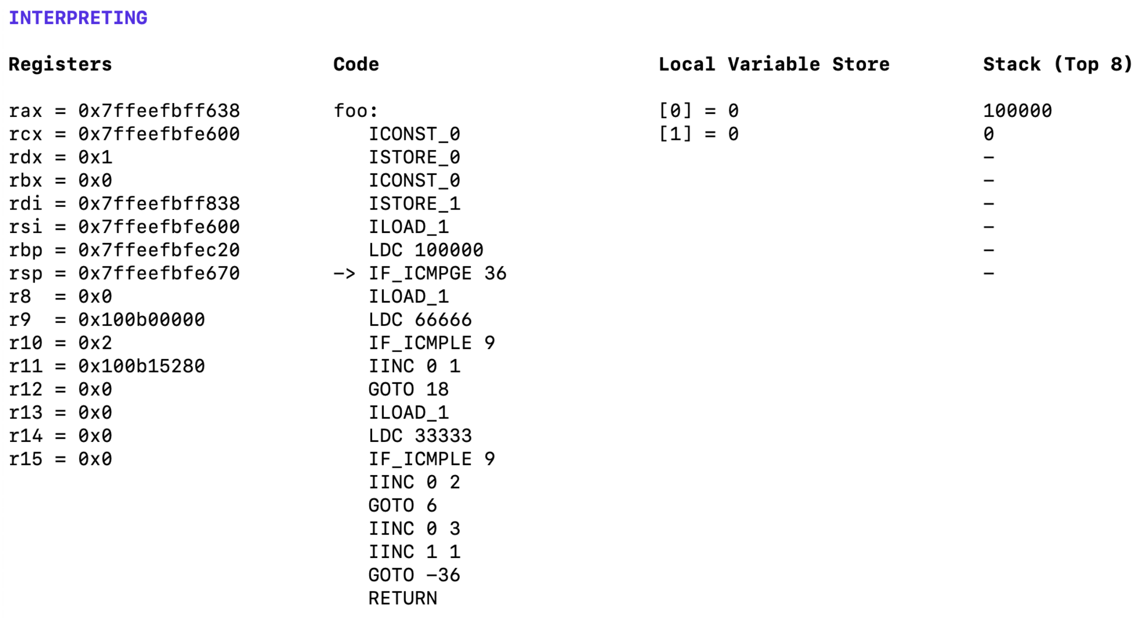
```
public static void foo() {
  int i = 0;
  for (int j = 0; j < 100000; j++) {
    if (j > 66666) {
      i += 1;
    } else if (j > 33333) {
      i += 2;
    } else {
      i += 3;
    }
  }
}
```

Figure 4.2: Example Java code which is run with the visualizer.

```
public static void foo();
  Code:
    0: ICONST_0
    1: ISTORE_0
    2: ICONST_0
    3: ISTORE_1
    4: ILOAD_1
    5: LDC 100000
    7: IF_ICMPGE 36
   10: ILOAD_1
   11: LDC 66666
   13: IF_ICMPLE 9
   16: IINC 0 1
   19: GOTO 18
   22: ILOAD_1
   23: LDC 33333
   25: IF_ICMPLE 9
   28: IINC 0 2
   31: GOTO 6
   34: IINC 0 3
   37: IINC 1 1
   40: GOTO -36
   43: RETURN
```

Figure 4.3: JVM byte-code of example in Figure 4.2. Note: Line numbers represent position in the byte-sequence and is therefore not purely sequential.

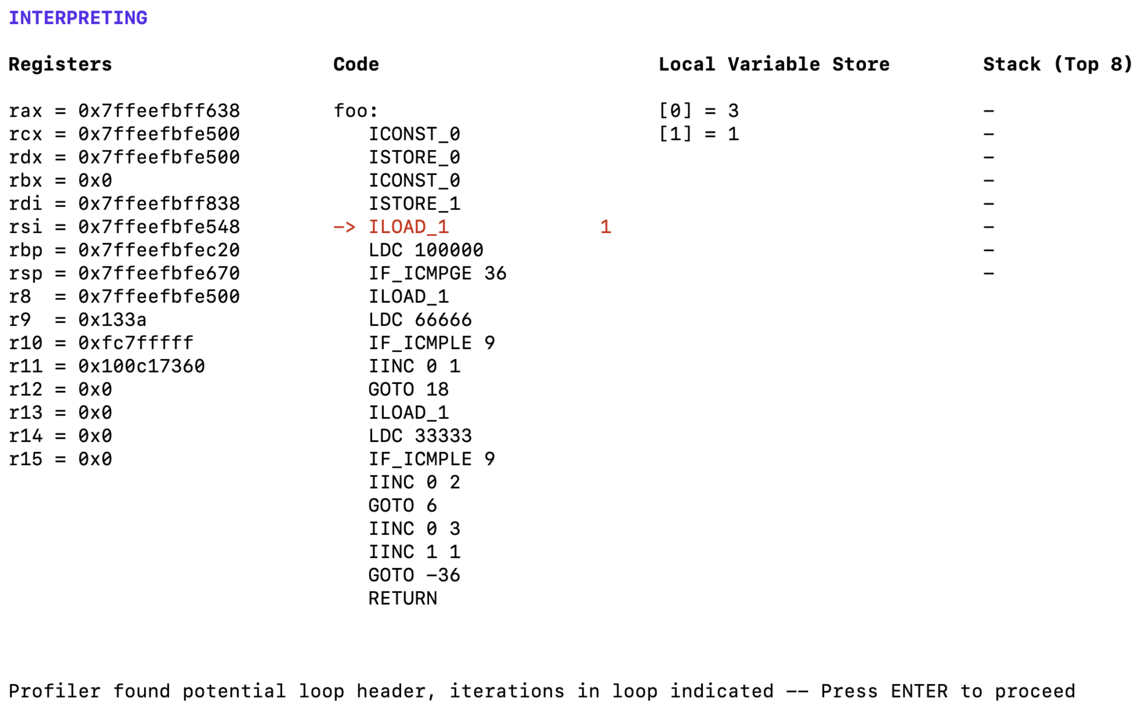
[1]Video example: `https://www.youtube.com/watch?v=v_Y4rq2IsPI`



```
INTERPRETING

Registers                  Code                       Local Variable Store       Stack (Top 8)

rax = 0x7ffeefbff638       foo:                       [0] = 0                    100000
rcx = 0x7ffeefbfe600          ICONST_0                [1] = 0                    0
rdx = 0x1                     ISTORE_0                                           -
rbx = 0x0                     ICONST_0                                           -
rdi = 0x7ffeefbff838          ISTORE_1                                           -
rsi = 0x7ffeefbfe600          ILOAD_1                                            -
rbp = 0x7ffeefbfec20          LDC 100000                                         -
rsp = 0x7ffeefbfe670       -> IF_ICMPGE 36                                       -
r8  = 0x0                     ILOAD_1
r9  = 0x100b00000             LDC 66666
r10 = 0x2                     IF_ICMPLE 9
r11 = 0x100b15280             IINC 0 1
r12 = 0x0                     GOTO 18
r13 = 0x0                     ILOAD_1
r14 = 0x0                     LDC 33333
r15 = 0x0                     IF_ICMPLE 9
                              IINC 0 2
                              GOTO 6
                              IINC 0 3
                              IINC 1 1
                              GOTO -36
                              RETURN
```

Figure 4.4: Visualizer showing interpretation of byte-code instruction.

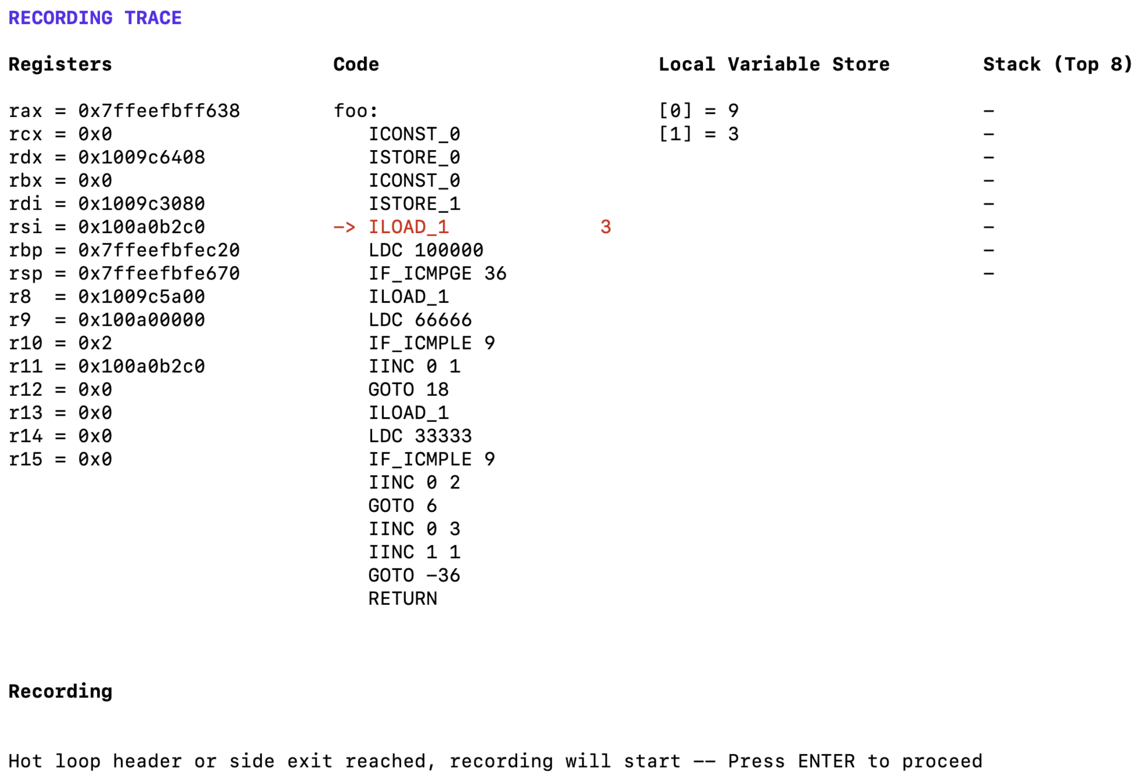
Figure 4.4 shows the visualizer while the tracing JIT compiler is performing pure byte-code interpretation. At this time, the visualizer presents some data to the viewer. First, the values contained in the general purpose registers, which in a sense is the lowest level representation of data in the compiler process. Also, we see the some byte-code instructions and an arrow indicating which instruction is to be evaluated next. This is to give the viewer an overview of program during execution. We also see the values placed in the local variable store as well as the top most values of the operand stack. In the top-left corner, we can see which state our tracing JIT compiler is currently in.



```
INTERPRETING

Registers                Code                     Local Variable Store     Stack (Top 8)

rax = 0x7ffeefbff638     foo:                     [0] = 3                  -
rcx = 0x7ffeefbfe500        ICONST_0              [1] = 1                  -
rdx = 0x7ffeefbfe500        ISTORE_0                                       -
rbx = 0x0                   ICONST_0                                       -
rdi = 0x7ffeefbff838        ISTORE_1                                       -
rsi = 0x7ffeefbfe548     -> ILOAD_1             1                          -
rbp = 0x7ffeefbfec20        LDC 100000                                     -
rsp = 0x7ffeefbfe670        IF_ICMPGE 36                                   -
r8  = 0x7ffeefbfe500        ILOAD_1
r9  = 0x133a                LDC 66666
r10 = 0xfc7fffff            IF_ICMPLE 9
r11 = 0x100c17360           IINC 0 1
r12 = 0x0                   GOTO 18
r13 = 0x0                   ILOAD_1
r14 = 0x0                   LDC 33333
r15 = 0x0                   IF_ICMPLE 9
                            IINC 0 2
                            GOTO 6
                            IINC 0 3
                            IINC 1 1
                            GOTO -36
                            RETURN


Profiler found potential loop header, iterations in loop indicated -- Press ENTER to proceed
```

Figure 4.5: Visualizer showing a possible hot loop header.

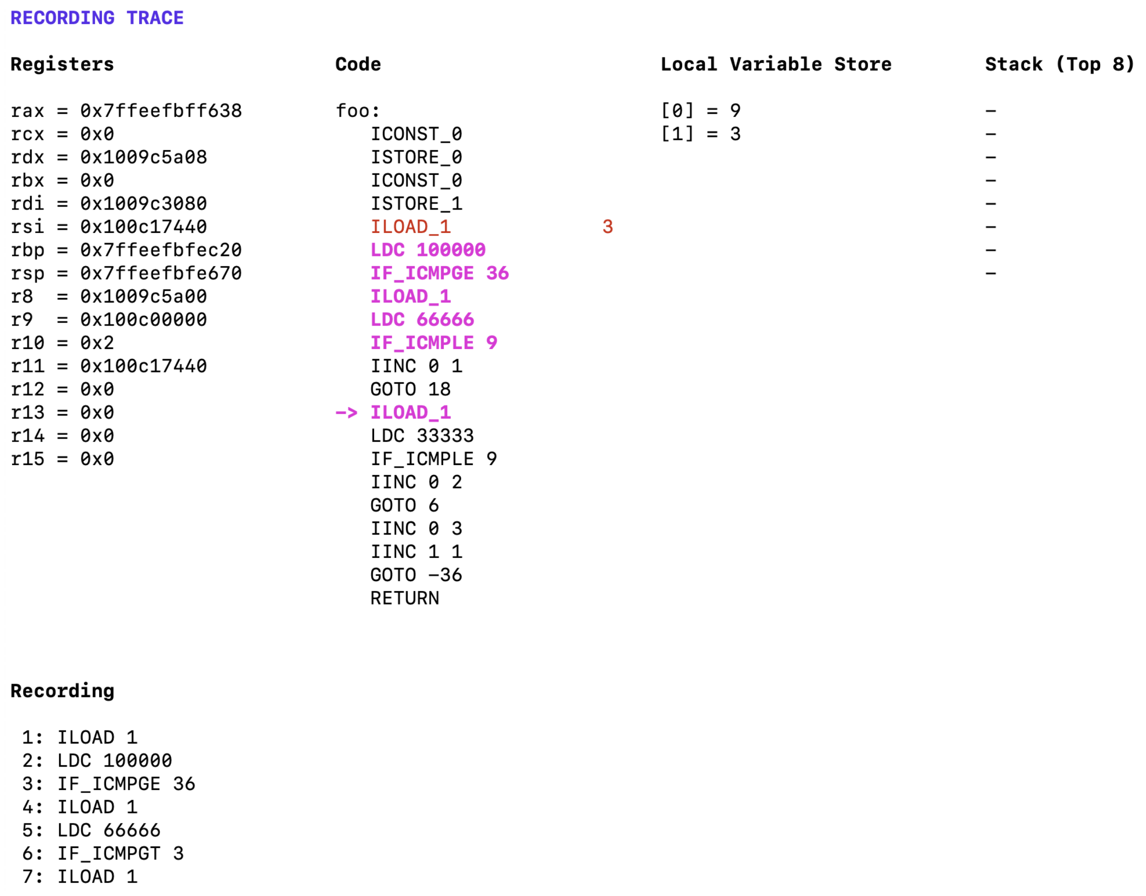
In Figure 4.5, we see that one line in the code has turned red and has a number after it. This means that the profiler has identified a loop header and started counting the number of iterations this loop has been run.



```
RECORDING TRACE

Registers                 Code                      Local Variable Store      Stack (Top 8)

rax = 0x7ffeefbff638      foo:                      [0] = 9                   -
rcx = 0x0                    ICONST_0               [1] = 3                   -
rdx = 0x1009c6408            ISTORE_0                                         -
rbx = 0x0                    ICONST_0                                         -
rdi = 0x1009c3080            ISTORE_1                                         -
rsi = 0x100a0b2c0         -> ILOAD_1           3                              -
rbp = 0x7ffeefbfec20         LDC 100000                                       -
rsp = 0x7ffeefbfe670         IF_ICMPGE 36                                     -
r8  = 0x1009c5a00            ILOAD_1
r9  = 0x100a00000            LDC 66666
r10 = 0x2                    IF_ICMPLE 9
r11 = 0x100a0b2c0            IINC 0 1
r12 = 0x0                    GOTO 18
r13 = 0x0                    ILOAD_1
r14 = 0x0                    LDC 33333
r15 = 0x0                    IF_ICMPLE 9
                             IINC 0 2
                             GOTO 6
                             IINC 0 3
                             IINC 1 1
                             GOTO -36
                             RETURN


Recording


Hot loop header or side exit reached, recording will start -- Press ENTER to proceed
```

Figure 4.6: Visualizer before recording of hot trace starts.

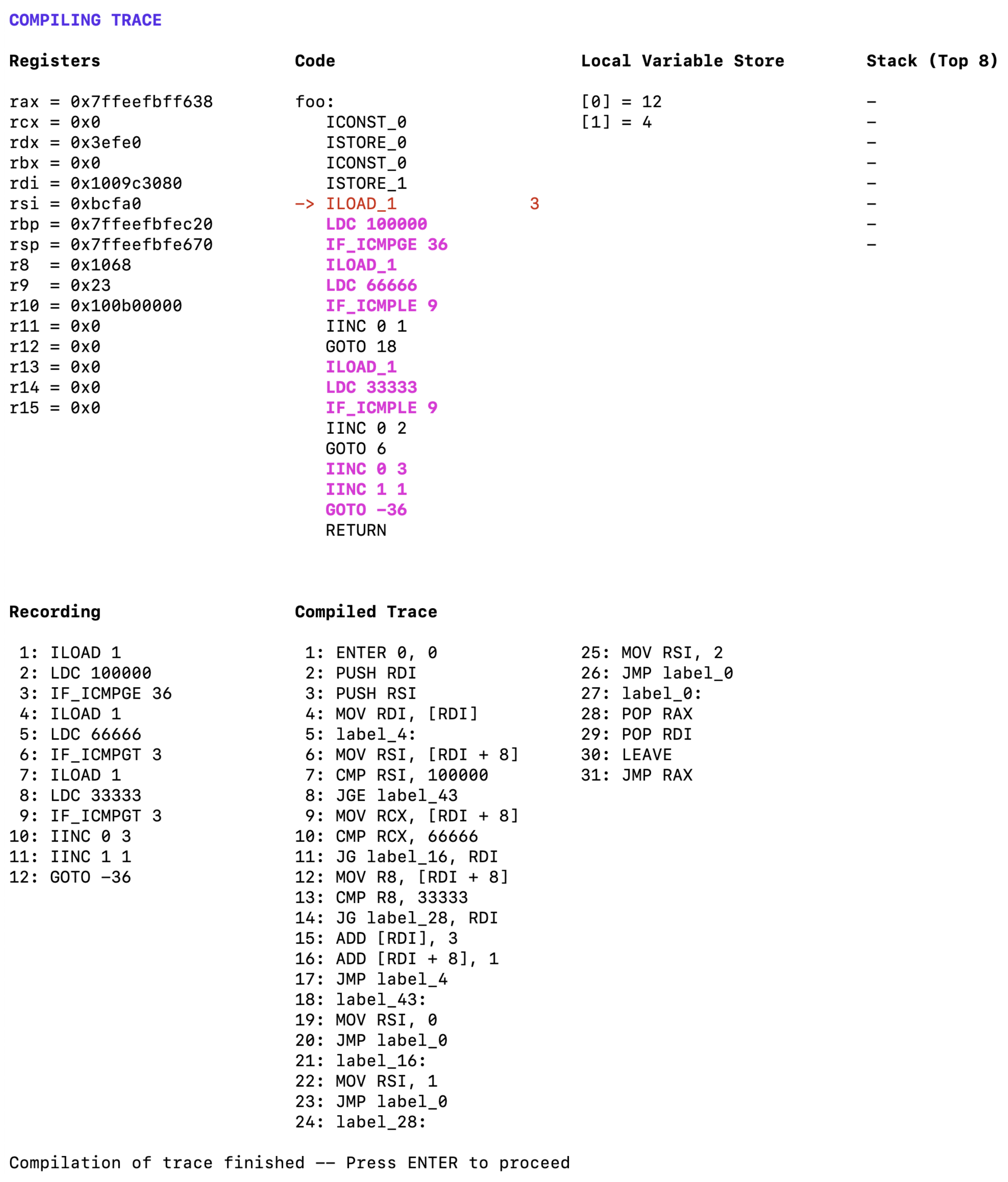
At the point shown in Figure 4.6 the loop header profiled in the previous step has been identified as hot and therefore recording will start. This is shown both by the blue text in the top-left corner and by the new column header below the registers.



```
RECORDING TRACE

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 9                     -
rcx = 0x0                      ICONST_0                 [1] = 3                     -
rdx = 0x1009c5a08              ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x1009c3080              ISTORE_1                                             -
rsi = 0x100c17440              ILOAD_1            3                                 -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x1009c5a00              ILOAD_1
r9  = 0x100c00000              LDC 66666
r10 = 0x2                      IF_ICMPLE 9
r11 = 0x100c17440              IINC 0 1
r12 = 0x0                      GOTO 18
r13 = 0x0                   -> ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Recording

 1: ILOAD 1
 2: LDC 100000
 3: IF_ICMPGE 36
 4: ILOAD 1
 5: LDC 66666
 6: IF_ICMPGT 3
 7: ILOAD 1
```

Figure 4.7: Visualizer during trace recording.

As the trace recorder records byte-codes, they turn purple in the visualizer to show that they are part of a trace. This is shown in Figure 4.7. We can see in the recorded instructions that some of them have changed, for example on line 4 where `ILOAD_1` becomes the parameterized version `ILOAD 1`. A more drastic change can be seen on line 6 in the recording, where we recorded the instruction `IF_ICMPLE` meaning to branch if one value is less than or equal to the other one, however the result is the instruction `IF_ICMPGT`. The new branch instruction is the opposite of the old one, the reasoning behind this is explained in Section 3.4. We can also clearly see that a trace does not need to be one contiguous block of code.

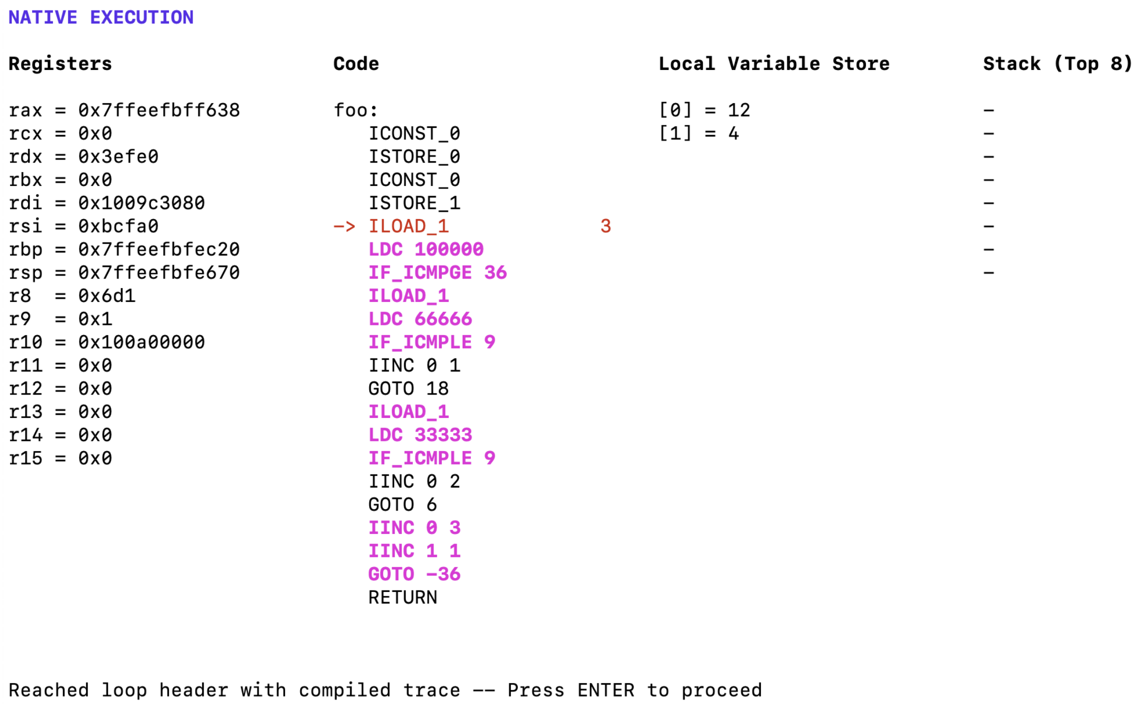


```
COMPILING TRACE

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 12                    -
rcx = 0x0                      ICONST_0                 [1] = 4                     -
rdx = 0x3efe0                  ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x1009c3080              ISTORE_1                                             -
rsi = 0xbcfa0               -> ILOAD_1            3                                 -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x1068                   ILOAD_1
r9  = 0x23                     LDC 66666
r10 = 0x100b00000              IF_ICMPLE 9
r11 = 0x0                      IINC 0 1
r12 = 0x0                      GOTO 18
r13 = 0x0                      ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Recording                   Compiled Trace

 1: ILOAD 1                  1: ENTER 0, 0              25: MOV RSI, 2
 2: LDC 100000               2: PUSH RDI                26: JMP label_0
 3: IF_ICMPGE 36             3: PUSH RSI                27: label_0:
 4: ILOAD 1                  4: MOV RDI, [RDI]          28: POP RAX
 5: LDC 66666                5: label_4:                29: POP RDI
 6: IF_ICMPGT 3              6: MOV RSI, [RDI + 8]      30: LEAVE
 7: ILOAD 1                  7: CMP RSI, 100000         31: JMP RAX
 8: LDC 33333                8: JGE label_43
 9: IF_ICMPGT 3              9: MOV RCX, [RDI + 8]
10: IINC 0 3                10: CMP RCX, 66666
11: IINC 1 1                11: JG label_16, RDI
12: GOTO -36                12: MOV R8, [RDI + 8]
                            13: CMP R8, 33333
                            14: JG label_28, RDI
                            15: ADD [RDI], 3
                            16: ADD [RDI + 8], 1
                            17: JMP label_4
                            18: label_43:
                            19: MOV RSI, 0
                            20: JMP label_0
                            21: label_16:
                            22: MOV RSI, 1
                            23: JMP label_0
                            24: label_28:

Compilation of trace finished -- Press ENTER to proceed
```

Figure 4.8: Visualizer showing compiled trace.

Figure 4.8 shows that the program counter is back at the start of the loop. Therefore, the trace recording is finished and the trace is compiled. Now we can see the entire recorded sequence of instructions and what the generated native code looks like. Interesting to note is that the actual native code corresponding to the recorded trace is only the lines between `label_4` and `JMP label_4`, more specifically between line 5 and 17, so much of the generated native code has to do with handling entering and exiting traces.

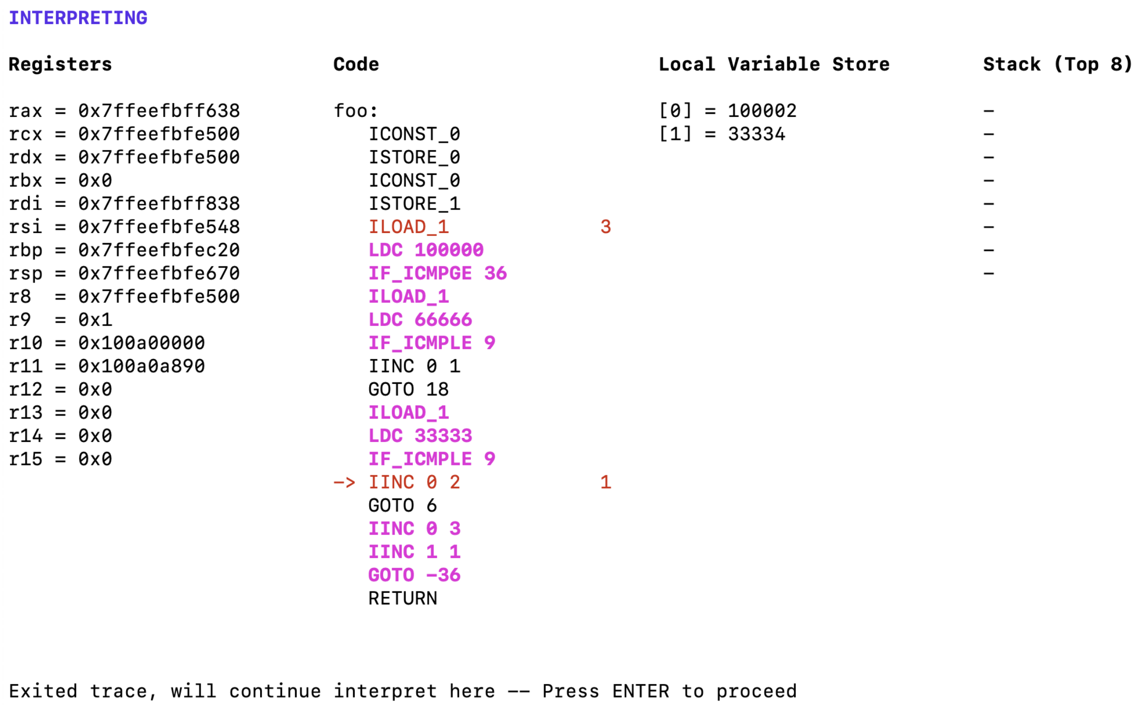
On line 6 which says `MOV RSI, [RDI + 8]` we put the value of a variable in the variable store into a register. As discussed in Section 3.8, we pass a list containing the values of the variables in the local variable store as an argument to the trace. The register RDI contains a reference to this list and therefore we can index it to retrieve the values.



```
NATIVE EXECUTION

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 12                    -
rcx = 0x0                      ICONST_0                 [1] = 4                     -
rdx = 0x3efe0                  ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x1009c3080              ISTORE_1                                             -
rsi = 0xbcfa0               -> ILOAD_1              3                               -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x6d1                    ILOAD_1
r9  = 0x1                      LDC 66666
r10 = 0x100a00000              IF_ICMPLE 9
r11 = 0x0                      IINC 0 1
r12 = 0x0                      GOTO 18
r13 = 0x0                      ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Reached loop header with compiled trace -- Press ENTER to proceed
```

Figure 4.9: Visualizer before executing native trace.

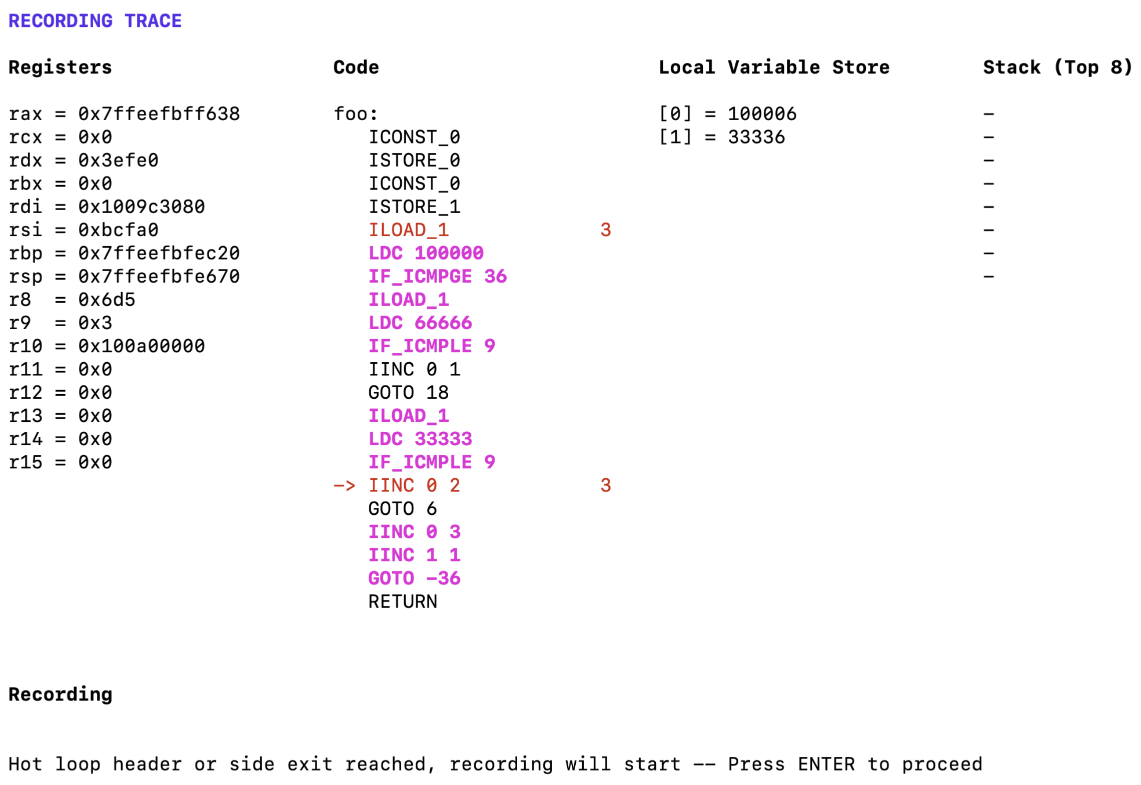
The recorded trace has now been compiled and placed into executable memory. Now when the interpreter continues, it detect that there is a native trace starting at its current position and will execute this trace. This is shown in Figure 4.9.



```
INTERPRETING

Registers                Code                     Local Variable Store     Stack (Top 8)

rax = 0x7ffeefbff638     foo:                     [0] = 100002             -
rcx = 0x7ffeefbfe500        ICONST_0              [1] = 33334              -
rdx = 0x7ffeefbfe500        ISTORE_0                                       -
rbx = 0x0                   ICONST_0                                       -
rdi = 0x7ffeefbff838        ISTORE_1                                       -
rsi = 0x7ffeefbfe548        ILOAD_1            3                           -
rbp = 0x7ffeefbfec20        LDC 100000                                     -
rsp = 0x7ffeefbfe670        IF_ICMPGE 36                                   -
r8  = 0x7ffeefbfe500        ILOAD_1
r9  = 0x1                   LDC 66666
r10 = 0x100a00000           IF_ICMPLE 9
r11 = 0x100a0a890           IINC 0 1
r12 = 0x0                   GOTO 18
r13 = 0x0                   ILOAD_1
r14 = 0x0                   LDC 33333
r15 = 0x0                   IF_ICMPLE 9
                         -> IINC 0 2           1
                            GOTO 6
                            IINC 0 3
                            IINC 1 1
                            GOTO -36
                            RETURN


Exited trace, will continue interpret here -- Press ENTER to proceed
```

Figure 4.10: Visualizer showing side exit which might become hot.

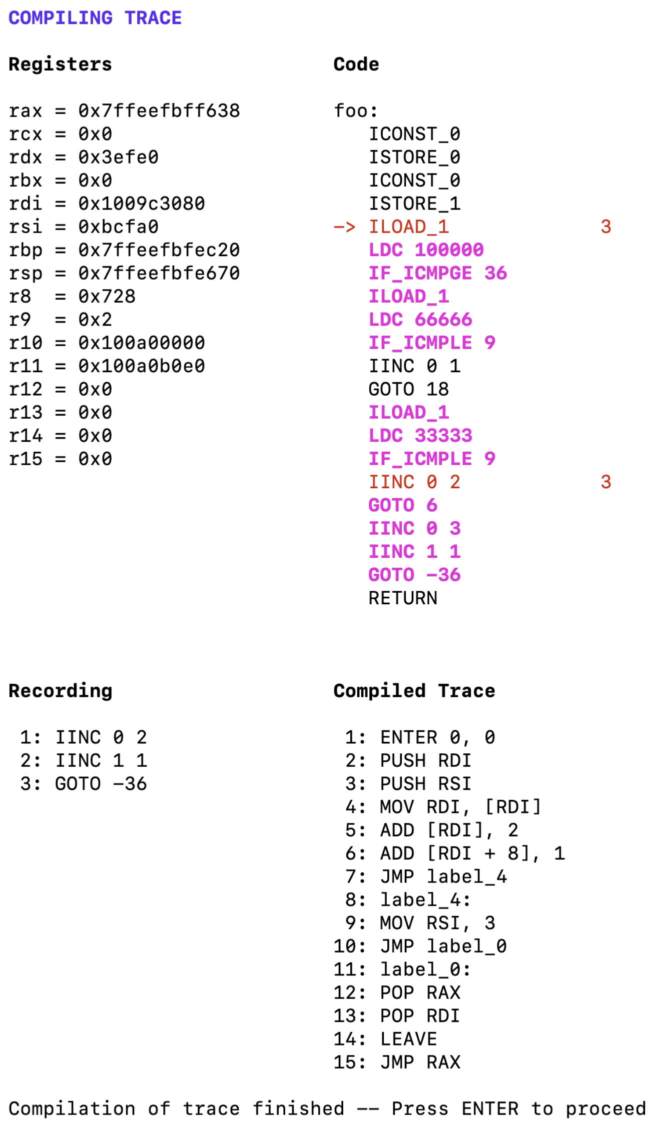
At this point, shown in Figure 4.10, we can see that we have exited the native trace. We have no way of visualizing when native code is executed so we rely on the fact that the variables in the local variable store have changed to see that we got the expected outcome from running the trace. As we want to be able to recognize hot side exits we can see that this exit is now being profiled by the profiler even though is not a loop header.



```
RECORDING TRACE

Registers                    Code                         Local Variable Store         Stack (Top 8)

rax = 0x7ffeefbff638         foo:                         [0] = 100006                 -
rcx = 0x0                       ICONST_0                  [1] = 33336                  -
rdx = 0x3efe0                   ISTORE_0                                               -
rbx = 0x0                       ICONST_0                                               -
rdi = 0x1009c3080               ISTORE_1                                               -
rsi = 0xbcfa0                   ILOAD_1             3                                  -
rbp = 0x7ffeefbfec20            LDC 100000                                             -
rsp = 0x7ffeefbfe670            IF_ICMPGE 36                                           -
r8  = 0x6d5                     ILOAD_1
r9  = 0x3                       LDC 66666
r10 = 0x100a00000               IF_ICMPLE 9
r11 = 0x0                       IINC 0 1
r12 = 0x0                       GOTO 18
r13 = 0x0                       ILOAD_1
r14 = 0x0                       LDC 33333
r15 = 0x0                       IF_ICMPLE 9
                             -> IINC 0 2            3
                                GOTO 6
                                IINC 0 3
                                IINC 1 1
                                GOTO -36
                                RETURN


Recording


Hot loop header or side exit reached, recording will start -- Press ENTER to proceed
```

Figure 4.11: Visualizer before recording hot side exit.

This is now the third time the trace has exited into this one side exit. As such, it is considered hot and we will start recording from where the exit starts. See Figure 4.11.

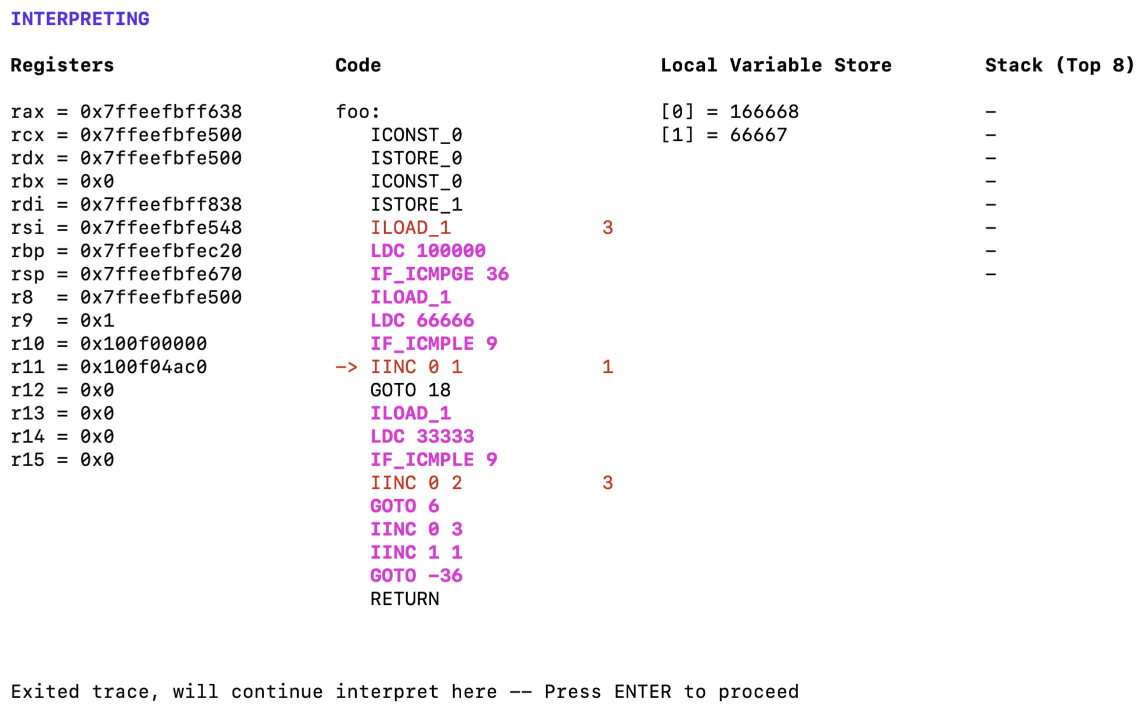


```
COMPILING TRACE

Registers                Code                     Local Variable Store     Stack (Top 8)

rax = 0x7ffeefbff638     foo:                     [0] = 100008             -
rcx = 0x0                   ICONST_0              [1] = 33337              -
rdx = 0x3efe0               ISTORE_0                                       -
rbx = 0x0                   ICONST_0                                       -
rdi = 0x1009c3080           ISTORE_1                                       -
rsi = 0xbcfa0            -> ILOAD_1           3                            -
rbp = 0x7ffeefbfec20        LDC 100000                                     -
rsp = 0x7ffeefbfe670        IF_ICMPGE 36                                   -
r8  = 0x728                 ILOAD_1
r9  = 0x2                   LDC 66666
r10 = 0x100a00000           IF_ICMPLE 9
r11 = 0x100a0b0e0           IINC 0 1
r12 = 0x0                   GOTO 18
r13 = 0x0                   ILOAD_1
r14 = 0x0                   LDC 33333
r15 = 0x0                   IF_ICMPLE 9
                            IINC 0 2          3
                            GOTO 6
                            IINC 0 3
                            IINC 1 1
                            GOTO -36
                            RETURN


Recording                Compiled Trace

 1: IINC 0 2              1: ENTER 0, 0
 2: IINC 1 1              2: PUSH RDI
 3: GOTO -36              3: PUSH RSI
                          4: MOV RDI, [RDI]
                          5: ADD [RDI], 2
                          6: ADD [RDI + 8], 1
                          7: JMP label_4
                          8: label_4:
                          9: MOV RSI, 3
                         10: JMP label_0
                         11: label_0:
                         12: POP RAX
                         13: POP RDI
                         14: LEAVE
                         15: JMP RAX

Compilation of trace finished -- Press ENTER to proceed
```

Figure 4.12: Visualizer after compiling side exit trace.

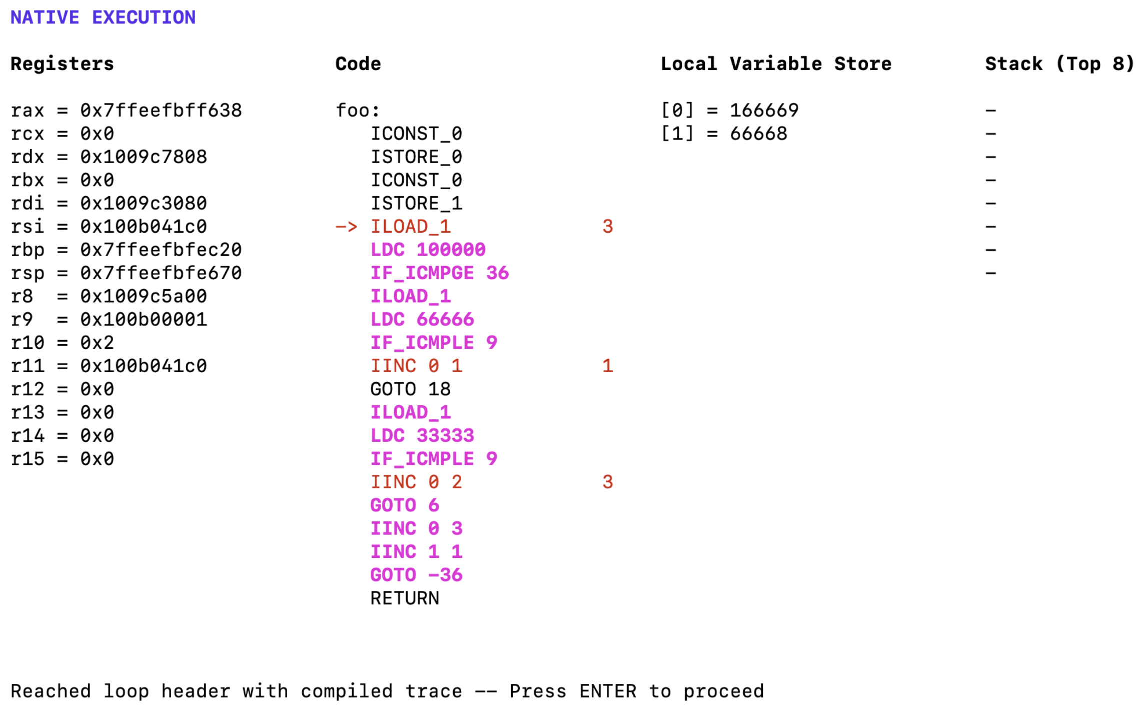
In Figure 4.12, we see that we have now recorded and compiled the side exit trace. Note that we finish recording not when the program counter reaches the point where recording started, but rather where the actual trace begins, which in this case is the start of the loop. One can see that the instruction `GOTO 6` is missing from the recording, this is because unconditional forward jumps has no function in a trace as discussed in Section 3.4.

Figures 4.13, 4.14 and 4.15 show the same steps as Figures 4.10, 4.11 and 4.12 for a second hot side exit.



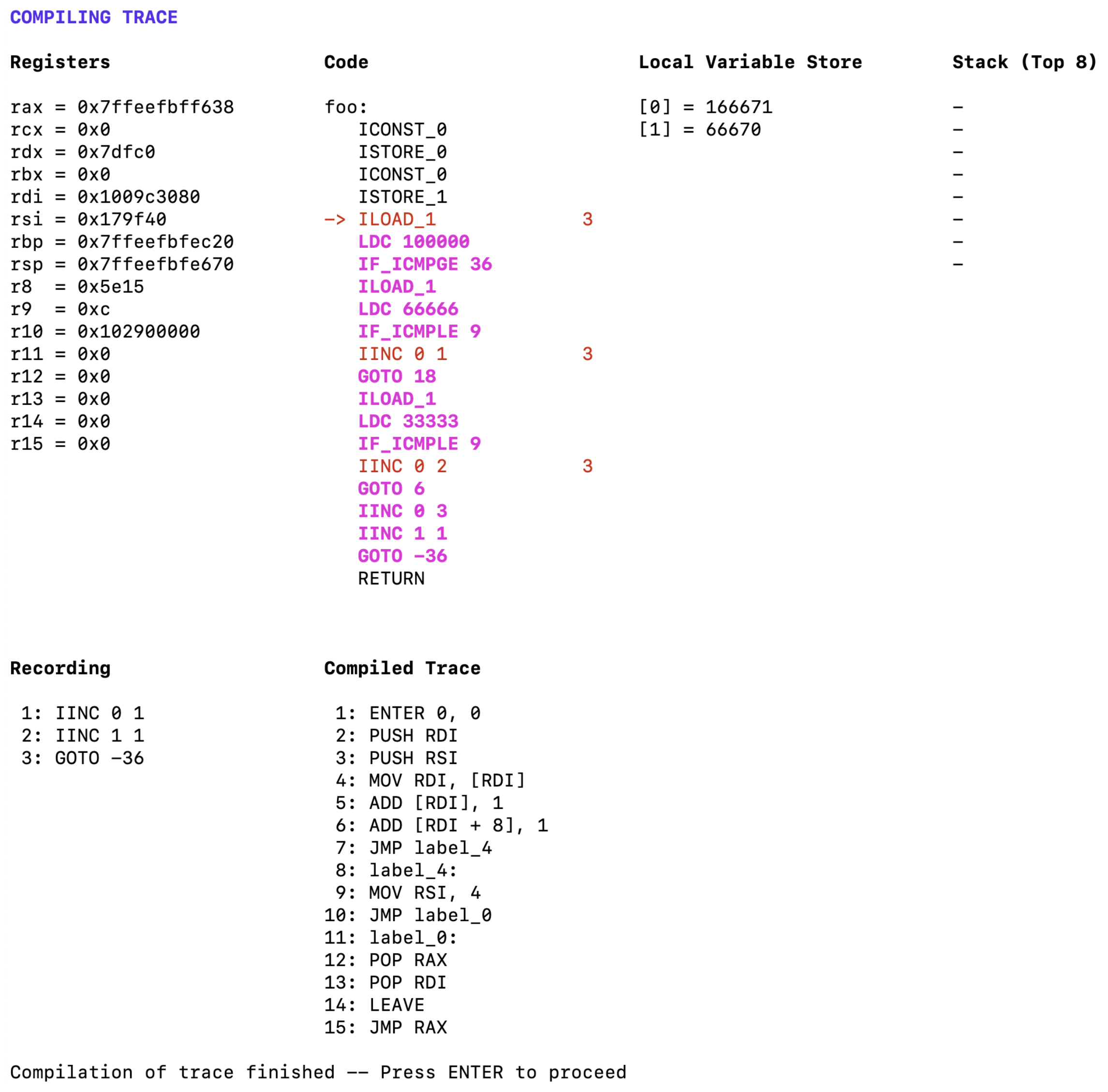
```
INTERPRETING

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 166668                -
rcx = 0x7ffeefbfe500           ICONST_0                 [1] = 66667                 -
rdx = 0x7ffeefbfe500           ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x7ffeefbff838           ISTORE_1                                             -
rsi = 0x7ffeefbfe548           ILOAD_1             3                                -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x7ffeefbfe500           ILOAD_1
r9  = 0x1                      LDC 66666
r10 = 0x100f00000              IF_ICMPLE 9
r11 = 0x100f04ac0           -> IINC 0 1            1
r12 = 0x0                      GOTO 18
r13 = 0x0                      ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2            3
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Exited trace, will continue interpret here -- Press ENTER to proceed
```

Figure 4.13: Visualizer showing second side exit which also might become hot.



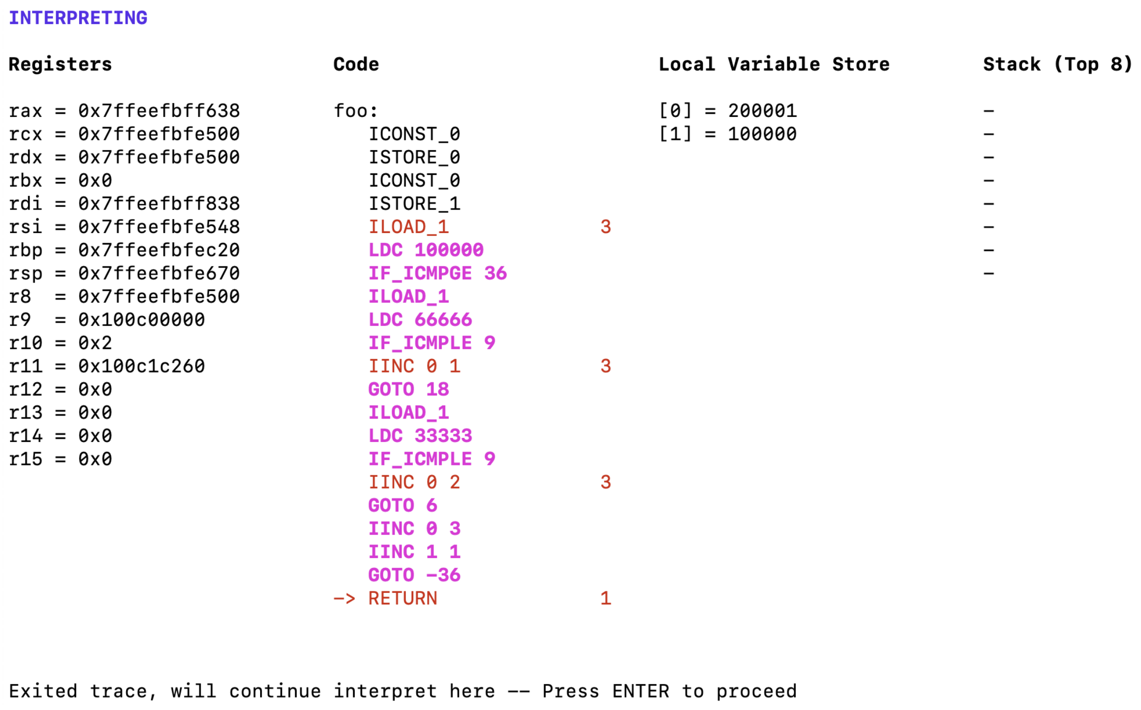
```
NATIVE EXECUTION

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 166669                -
rcx = 0x0                      ICONST_0                 [1] = 66668                 -
rdx = 0x1009c7808              ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x1009c3080              ISTORE_1                                             -
rsi = 0x100b041c0           -> ILOAD_1             3                                -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x1009c5a00              ILOAD_1
r9  = 0x100b00001              LDC 66666
r10 = 0x2                      IF_ICMPLE 9
r11 = 0x100b041c0              IINC 0 1            1
r12 = 0x0                      GOTO 18
r13 = 0x0                      ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2            3
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Reached loop header with compiled trace -- Press ENTER to proceed
```

Figure 4.14: Visualizer before recording second hot side exit.



```
COMPILING TRACE

Registers                   Code                        Local Variable Store        Stack (Top 8)

rax = 0x7ffeefbff638        foo:                        [0] = 166671                -
rcx = 0x0                      ICONST_0                 [1] = 66670                 -
rdx = 0x7dfc0                  ISTORE_0                                             -
rbx = 0x0                      ICONST_0                                             -
rdi = 0x1009c3080              ISTORE_1                                             -
rsi = 0x179f40              -> ILOAD_1               3                              -
rbp = 0x7ffeefbfec20           LDC 100000                                           -
rsp = 0x7ffeefbfe670           IF_ICMPGE 36                                         -
r8  = 0x5e15                   ILOAD_1
r9  = 0xc                      LDC 66666
r10 = 0x102900000              IF_ICMPLE 9
r11 = 0x0                      IINC 0 1              3
r12 = 0x0                      GOTO 18
r13 = 0x0                      ILOAD_1
r14 = 0x0                      LDC 33333
r15 = 0x0                      IF_ICMPLE 9
                               IINC 0 2              3
                               GOTO 6
                               IINC 0 3
                               IINC 1 1
                               GOTO -36
                               RETURN


Recording                   Compiled Trace

 1: IINC 0 1                 1: ENTER 0, 0
 2: IINC 1 1                 2: PUSH RDI
 3: GOTO -36                 3: PUSH RSI
                             4: MOV RDI, [RDI]
                             5: ADD [RDI], 1
                             6: ADD [RDI + 8], 1
                             7: JMP label_4
                             8: label_4:
                             9: MOV RSI, 4
                            10: JMP label_0
                            11: label_0:
                            12: POP RAX
                            13: POP RDI
                            14: LEAVE
                            15: JMP RAX

Compilation of trace finished -- Press ENTER to proceed
```

Figure 4.15: Visualizer after compiling second side exit.



```
INTERPRETING

Registers                Code                     Local Variable Store     Stack (Top 8)

rax = 0x7ffeefbff638     foo:                     [0] = 200001             -
rcx = 0x7ffeefbfe500        ICONST_0              [1] = 100000             -
rdx = 0x7ffeefbfe500        ISTORE_0                                       -
rbx = 0x0                   ICONST_0                                       -
rdi = 0x7ffeefbff838        ISTORE_1                                       -
rsi = 0x7ffeefbfe548        ILOAD_1           3                            -
rbp = 0x7ffeefbfec20        LDC 100000                                     -
rsp = 0x7ffeefbfe670        IF_ICMPGE 36                                   -
r8  = 0x7ffeefbfe500        ILOAD_1
r9  = 0x100c00000           LDC 66666
r10 = 0x2                   IF_ICMPLE 9
r11 = 0x100c1c260           IINC 0 1          3
r12 = 0x0                   GOTO 18
r13 = 0x0                   ILOAD_1
r14 = 0x0                   LDC 33333
r15 = 0x0                   IF_ICMPLE 9
                            IINC 0 2          3
                            GOTO 6
                            IINC 0 3
                            IINC 1 1
                            GOTO -36
                         -> RETURN            1


Exited trace, will continue interpret here -- Press ENTER to proceed
```

Figure 4.16: Visualizer showing the interpreter taking back control after a trace has finished executing.

Figure 4.16 shows the visualizer after exiting the trace when the loop has finished. We can see that the entire loop has been compiled while the code before and after the loop, which was only run once, has not been compiled.

5
Experiments

In this chapter, we discuss the result of some experiments carried out to test the performance of our tracing JIT compiler. In order to have something to compare against, we evaluate the programs in three different ways: (1) pure byte-code interpretation using our JVM interpreter, (2) our tracing JIT compiler and (3) using OpenJDK 15 [20] as a base line. We use OpenJDK's Java compiler to generate the class files of the test programs and all timings are the average over 1000 test runs.

5.1 Extracting Real Execution Time

Compared to our compiler, OpenJDK's JVM has a notoriously long start-up time. This means that comparing the total running time of the Java process to our compilers process while evaluating a program would be highly misleading. In order to calculate the actual execution time of OpenJDK, we extract a log file generated at run-time. This log contains the tasks performed by the JVM accompanied with timestamps. We are interested in two timestamps: the first when initialization of the JVM is complete and the actual test file is loaded, the second when the JVM starts its shutdown process. We consider the difference between these two timestamps to be the actual execution time of the program.

Note that there is no one correct way of extracting the execution time of the JVM so this is simply a heuristic used to try to get a fair comparison.

Measuring the execution time of our compiler involves measuring the time spent in the main loop in the code in Figure 3.2. This means we disregard the time spent parsing the program and only consider the actual execution time. For the sake of comparison, parsing a program the size of the test-cases takes around five to six milliseconds.

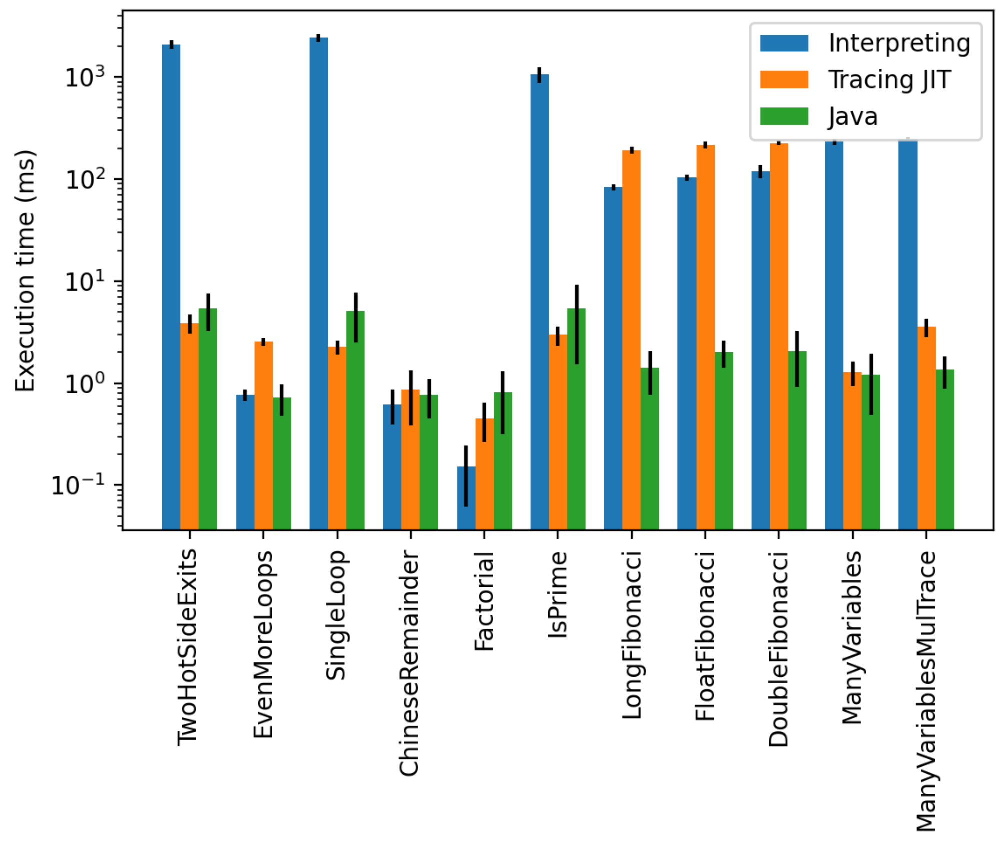


Figure 5.1: Mean execution time in milliseconds of a few test cases, lower is better. The black lines on each bar represents the standard deviation for that test. Note that the y-axis uses a logarithmic scale meaning each ”step” represents a tenfold increase in execution time. Exact values and more information can be found in Table 5.1.

Test name	**SLOC**	**Interpreter**	**Tracing**	**Java**	**#Traces**	**Traces size**
TwoHotSideExits	15	2092.4ms	3.9ms	5.4ms	3	195B
EvenMoreLoops	53	0.8ms	2.6ms	0.7ms	15	1677B
SingleLoop	10	2422.6ms	2.3ms	5.1ms	1	73B
ChineseRemainder	34	0.6ms	0.9ms	0.8ms	1	149B
Factorial	11	0.2ms	0.5ms	0.8ms	1	65B
IsPrime	21	1061.6ms	3.0ms	5.4ms	2	193B
LongFibonacci	11	83.0ms	191.5ms	1.4ms	0	0B
FloatFibonacci	11	103.3ms	216.8ms	2.0ms	0	0B
DoubleFibonacci	11	119.5ms	226.3ms	2.1ms	0	0B
ManyVariables	30	231.1ms	1.3ms	1.2ms	1	449B
ManyVariablesMulTrace	40	249.6ms	3.6ms	1.4ms	4	1936B

Table 5.1: Exact measurements from tests carried out. SLOC stands for ”Source lines of code” and is a measure for the actual number of executable lines of code in a file.

5.2 Tests

In this section, we provide a detailed explanation of the different test programs which produced the results shown in Figure 5.1 and Table 5.1.

5.2.1 Simple Looping Programs

```java
public class TwoHotSideExits {
  public static void main(String[] args) {
    int i = 0;
    for (int j = 0; j < 300000; j++) {
      if (j < 100000) {
        i += 1;
      } else if (j < 200000) {
        i += 2;
      } else {
        i += 3;
      }
    }
    System.out.println(i);
  }
}
```

Figure 5.2: Test case: TwoHotSideExits.

```java
public class SingleLoop {
  public static void main(String[] args) {
    int j = 1;
    for (int i = 0; i < 500000; i++) {
      int k = i + j;
      j = k;
    }
    System.out.println(j);
  }
}
```

Figure 5.3: Test case: SingleLoop.

The tests `TwoHotSideExits`, `SingleLoop` and `IsPrime` (seen in Figures 5.2, 5.3 and 5.4) are examples of where the power of tracing JIT compilation is clearly shown. The feature these programs have in common is that they contain loops that run for many iterations. As such, most of the iterations are spent executing native code; this is also true for when the code branches and we get several traces as in `TwoHotSideExits`. These examples also clearly show that pure byte-code interpretation is not viable in the long run.

```java
public class IsPrime {
  public static void main(String[] args) {
    int potentialPrime = 104729;
    int largestCheck = integerRoot(potentialPrime);
    for (int i = 2; i < largestCheck; i++) {
      if (potentialPrime % i == 0) {
        System.out.println(0);
        return;
      }
    }
    System.out.println(1);
  }

  public static int integerRoot(int num) {
    for (int i = 1; i < num; i++) {
      if (num % i == i) {
        return i;
      }
    }
    return num;
  }
}
```

Figure 5.4: Test case: IsPrime.

5.2.2 Tracing JIT Compilation Pitfalls

```java
public class Factorial {
  public static void main(String[] args) {
    int f = factorial(12);
    System.out.println(f);
  }

  public static int factorial(int n) {
    int accumulator = 1;
    for (int i = 2; i <= n; i++) accumulator *= i;
    return accumulator;
  }
}
```

Figure 5.5: Test case: Factorial.

There are pitfalls for tracing JIT compilers where some loops are encountered just enough times to be considered hot but not much more. In this case, a lot of time is spent compiling code but no real benefit is gained from it as the generated native code is run very little or not at all. Examples of this can be seen in the tests `EvenMoreLoops` (located in Appendix B due to its size), `ChineseRemainder` and `Factorial` seen in Figures B.1, 5.6 and 5.5.

`EvenMoreLoops` is a test case designed specifically to be a trap for this kind of behavior as every loop is run enough times to be compiled into native code but then

```java
public class ChineseRemainder {
  public static long mul_inv(int a, int b) {
    int b0 = b, t, q;
    int x0 = 0, x1 = 1;
    if (b == 1) return 1;
    while (a > 1) {
      q = a / b;
      t = b;
      b = a % b;
      a = t;
      t = x0;
      x0 = x1 - q * x0;
      x1 = t;
    }
    if (x1 < 0) x1 += b0;
    return x1;
  }
  public static int chinese_remainder(int n1, int n2, int n3, int
   a1, int a2, int a3) {
    int p, i, prod = 1, sum = 0;
    prod = n1 * n2 * n3;
    p = prod / n1;
    sum += a1 * mul_inv(p, n1) * p;
    p = prod / n2;
    sum += a2 * mul_inv(p, n2) * p;
    p = prod / n3;
    sum += a3 * mul_inv(p, n3) * p;
    return ((sum % prod) + prod) % prod;
  }

  public static void main(String[] args) {
    System.out.println(chinese_remainder(997, 991, 983, 123, 14,
   66));
  }
}
```

Figure 5.6: Test case: ChineseRemainder.

never run again after that. However, the other two are examples of code that could be found in a real setting. Both `ChineseRemainder` and `Factorial` contain loops that are executed as native code but run for too few iterations to make up for the compilation time.

5.2.3 Exploding Memory Usage Due to Code Duplication

```java
public class ManyVariables {
  public static void main(String[] args) {
    int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t;
    a = b = c = d = e = f = g = h = i = j = k = l = m = n = o = p =
    q = r = s = t = 0;
    for (int index = 0; index < 5000; index++) {
      a += index; b += index; c += index; d += index;
      e += index; f += index; g += index; h += index;
      i += index; j += index; k += index; l += index;
      m += index; n += index; o += index; p += index;
      q += index; r += index; s += index; t += index;
    }
    System.out.println(
        a + b + c + d + e + f + g + h + i + j + k + l + m + n + o +
    p + q + r + s + t);
  }
}
```

Figure 5.7: Test case: ManyVariables.

```java
public class ManyVariablesMulTrace {
  public static void main(String[] args) {
    int a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t;
    a = b = c = d = e = f = g = h = i = j = k = l = m = n = o = p =
    q = r = s = t = 0;
    int offset;
    for (int index = 0; index < 5000; index++) {
      if (index % 2 == 0) {
        offset = 1;
      } else if (index % 3 == 0) {
        offset = 2;
      } else if (index % 5 == 0) {
        offset = 3;
      } else {
        offset = 4;
      }
      a += offset; b += offset; c += offset; d += offset;
      e += offset; f += offset; g += offset; h += offset;
      i += offset; j += offset; k += offset; l += offset;
      m += offset; n += offset; o += offset; p += offset;
      q += offset; r += offset; s += offset; t += offset;
    }
    System.out.println(
        a + b + c + d + e + f + g + h + i + j + k + l + m + n + o +
    p + q + r + s + t);
  }
}
```

Figure 5.8: Test case: ManyVariablesMulTrace.

Table 5.1 shows, apart from execution times, how many traces are compiled and how much memory these allocate during run-time. The tests `ManyVariables` and `ManyVariableMulTrace` contain many variables inside of a loop which produces fairly large traces. `ManyVariableMulTrace` starts with a few if-statements which only change the value to be added to all variables, a simplified illustration of this program can be seen in Figure 5.9. Since each path through the loop is compiled separately, the structure of the generated native code is as shown in Figure 5.10. We notice a problem here where this type of program produces a lot of duplicated code when tracing, this can also be seen in Table 5.1 where the size of the generated code is calculated to be about four times larger for this test than for `ManyVariables` even though the source programs are similar.

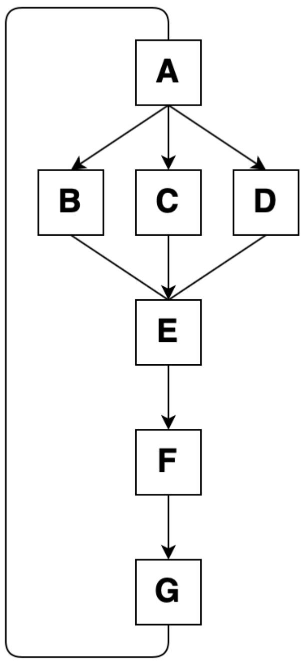


Figure 5.9: How the Java byte-code is structured.

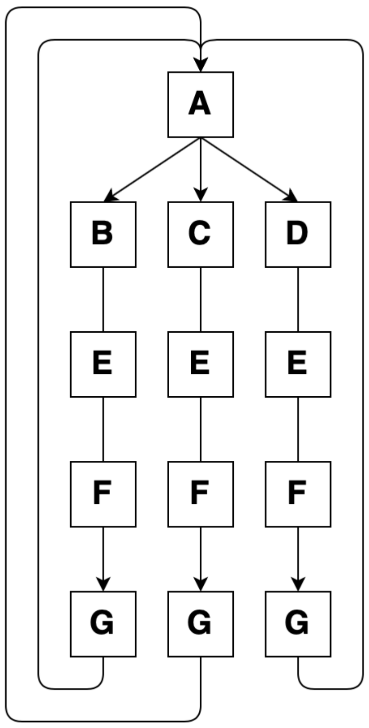


Figure 5.10: How the generated code is structured.

5.2.4 Interpreter Fallback

`LongFibonacci`, `FloatFibonacci` and `DoubleFibonacci` are presented to show two things. First, at this point our compiler does not trace recursive functions. So for these three test cases, we show that our compiler can fall back to interpretation of the programs instead. Second, these tests show our interpreter's capability to handle different types. Similar to the recursive element in these tests, our byte-code-to-native-code compiler currently doesn't support types other than `int`.

```java
public class LongFibonacci {
  public static void main(String[] args) {
    System.out.println(fibonacci(20));
  }
  public static long fibonacci(long n) {
    if (n <= 2) return 1;
    else return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Figure 5.11: Test case: LongFibonacci.

```java
public class FloatFibonacci {
  public static void main(String[] args) {
    System.out.println(fibonacci(20f));
  }
  public static float fibonacci(float n) {
    if (n <= 2f) return 1f;
    else return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Figure 5.12: Test case: FloatFibonacci.

```java
public class DoubleFibonacci {
  public static void main(String[] args) {
    System.out.println(fibonacci(20.0));
  }
  public static double fibonacci(double n) {
    if (n <= 2.0) return 1.0;
    else return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Figure 5.13: Test case: DoubleFibonacci.

6 Discussion

In this chapter we reflect on whether we achieved our goal of developing an understandable tracing JIT compiler. The chapter also contains some thoughts on possible future work.

6.1 Understandability

The goal of understandability is difficult to measure objectively. As a result, we need some other way of reasoning about whether we achieved this goal. We will describe the choices we have made that we think contribute to the understandability as well as comparing this project to other tracing JIT compilers.

6.1.1 Designed for Understandability

When implementing our tracing JIT compiler, an important step was to identify the core modules representing the different functionalities and implementing them as independently as possible from each other. One effect of this is that we can easily abstract the functionality of each module and have one central point which transfers control of the program between these modules (the run-time loop in Figure 3.2). Another effect is that each module can be developed separately so that, e.g., improving the compiler has no effect on how the interpreter or profiler operates. In theory, we could even switch out the byte-code-to-native-code compiler to an entirely different one, provided that its input- and output protocol stays the same.

We have made sure that the run-time loop in Figure 3.2, the diagram in Figure 3.1 and the description in Section 3.1 are equivalent, i.e., they describe the exact same thing in different ways. This can be helpful as it describes the same concepts from different perspectives so that one does not miss out on any information if one of the perspectives does not fit the reader.

The visualizing tool we have developed, described in Chapter 4, can be used to further ones understanding of what goes on at every step of the compiler's run-time loop. This tool removes some of the ”black box”-ness of the tracing JIT compiler as it shows some of its otherwise hidden functionality. Most notably it shows which parts of the code are being compiled and when, it can also show the current evaluation step and display the results from, e.g., recording and compiling a trace.

Many of the concepts described here which we believe contribute to understandability also apply to best practices in software engineering. Especially separation of concern and modularity which are popular design patterns used in large scale projects to increase their maintainability and to provide more freedom when working on specific modules as less time can be spent changing other code not directly connected to the module being worked on.

When investigating other tracing JIT compilers, we were unable to find examples of a clear central point in the code similar to the one described in this section. Nor were we able to find any tools for visualizing how the compiler operates during evaluation of a program.

6.1.2 Comparisons to Related Work

This section discusses four examples of projects on the same theme as the one described in this thesis: TraceMonkey, HotPathVM, Dynamo and Tamarin-Tracing. The discussion will cover the unique goals and achievements of the different projects.

The tracing JIT compiler (**TraceMonkey**) formerly used in Mozilla's JavaScript engine (SpiderMonkey) is probably the most prominent usage of tracing JIT compilers. The paper on TraceMonkey by Gal et al. [12] focuses on the tracing JIT compiler's ability to compile and run the dynamically typed language JavaScript. Dynamically typed languages comes with a new problem, namely that each combination of types that variables take requires its own trace. This further complicates the compiler as where our implementation can only have a single trace per loop, TraceMonkey can have arbitrarily many depending on how many variables are used and how many different types they assume. The paper is highly technical and focuses purely on performance and functionality. A comparison between TraceMonkey and our understandable tracing JIT would not be entirely fair since SpiderMonkey is a complete JavaScript engine rather than a stand-alone tracing JIT compiler so naturally it is a larger and more interconnected system. In other words, understandability was not a core aim of TraceMonkey.

HotPathVM is a tracing JIT compiler, extending the functionality of a Java Virtual Machine for embedded devices [11]. As such, the focus of this compiler is to make it as small and lightweight as possible. The sole objective of the HotPathVM project was to increase performance while maintaining a small size, which introduces another level of complexity to the project.

Dynamo seems to be the earliest example of trace-based optimizations [1]. In contrast to other tracing JIT compilers, Dynamo interprets native machine code instead of byte-code instructions, optimizing the trace of native instructions rather than generating native code. In an attempt to describe how Dynamo operates, the authors provide an architecture diagram of their system (See Figure 6.1). When studying this figure, one can see some similarities between it and Figure 3.1. For example the state description, "lookup branch target in cache" from Dynamo describes the same process as "Is there a compiled trace starting at current PC?" from our diagram. Figures like this contribute to helping the reader understand this complex system, even though understandability is not an explicit goal of the paper.

Tamarin-Tracing is a tracing JIT compiler for Adobe's ActionScript [13]. It is similar to TraceMonkey but instead uses the *Forth* byte-code language as an intermediate representation. The paper by Chang et al. provides a high level theoretical descriptions of the workings of their tracing JIT compiler.

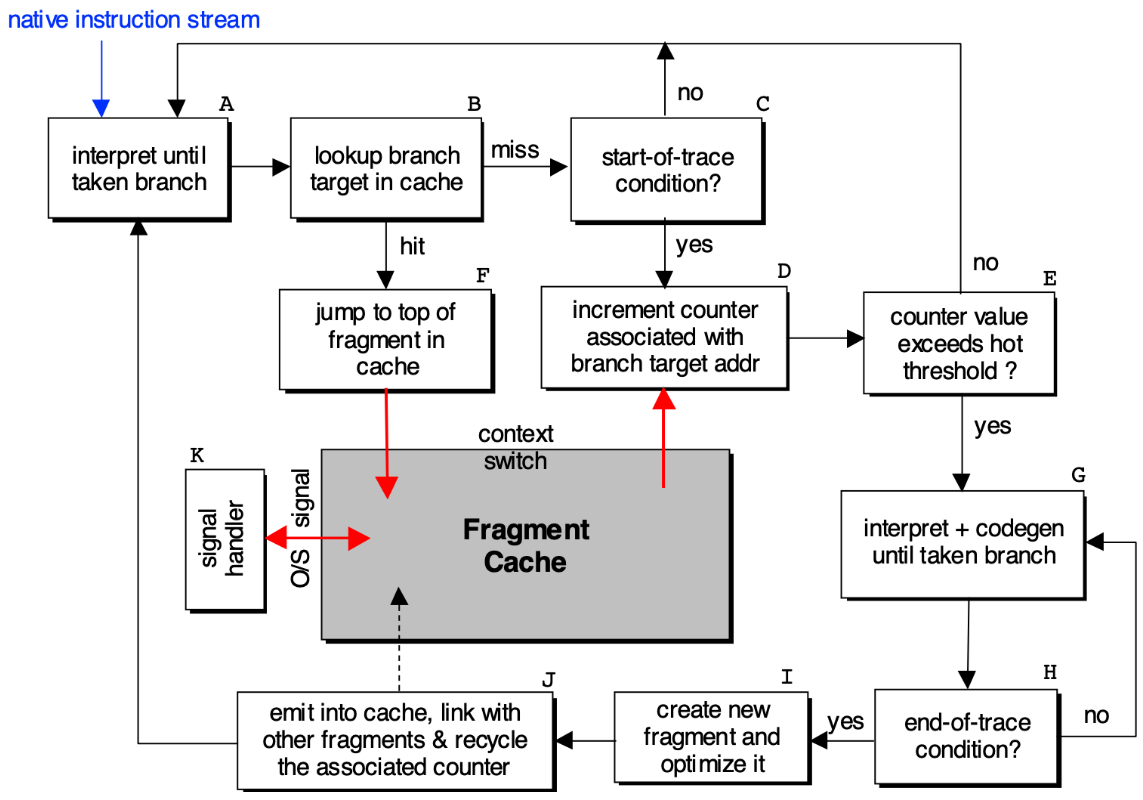

native instruction stream
A
interpret until taken branch
B
lookup branch target in cache
miss
C
start-of-trace condition?
no
hit
yes
F
jump to top of fragment in cache
D
increment counter associated with branch target addr
no
E
counter value exceeds hot threshold ?
yes
context switch
K
signal handler
O/S signal
Fragment Cache
G
interpret + codegen until taken branch
J
emit into cache, link with other fragments & recycle the associated counter
I
create new fragment and optimize it
yes
H
end-of-trace condition?
no

Figure 6.1: Control flow of Dynamo. From Bala et al. [1]. Reproduced with permission.

While Tamarin-Tracing has a clear description and Dynamo has an architecture diagram, which both help the reader understand the core concepts of a tracing JIT compiler, our thesis include both of these elements as well as a code example to make it even more clear. To our knowledge at the time of this writing, this is the first example of providing several different descriptions of the same tracing JIT compiler.

6.2 Results

Even though performance was not a stated goal of this project our compiler still performed well in some test cases. It seems that when the code contains very hot loops, i.e., loops with many iterations, our compiler actually seems to outperform even OpenJDK. This is without performing any optimizations on the generated code other than tracing JIT compilation.

Our interpreter is very slow. This is most notable in the tests where tracing JIT compilation performs very well. Since the interpreter is still part of the tracing JIT compiler, improving the performance of the interpreter would lead to better results.

Even though executing non-optimized native code is much faster than executing byte-codes directly, there is still room for improvement for the native code generation.

6.3 Future Work

Due to the time constraint of this thesis project, we chose to leave out some features, in this section we will discuss some of these features and explain how they could be implemented in a future version of this project.

6.3.1 Optimizing the Compiler

While compiling there's room for optimization to be made. When compiling AOT, one can spend quite a long time optimizing the code as the end result is more important than the running time of the compiler. This can create very high quality and efficient native code. In contrast, when using a JIT compiler the time spent compiling must be made up by the faster running code. As such, very time consuming optimizations might not be suitable as the extra efficiency of the end result does not make up for the time it takes to perform these optimizations.

Pure byte-code interpretation is very slow and early tests show significant improvement in evaluation time when performing tracing JIT compilation without any further optimizations. Even so, there are some optimization techniques that may be worth considering implementing.

6.3.1.1 Single Static Assignment Form

To simplify many types of compiler optimization, one can transform the recorded byte-code instructions into single static assignment (SSA) form. SSA form is an intermediate representation of code in which each variable is assigned a value only once. SSA form is useful for compilation as it makes the actual compilation faster and it opens up more opportunities for optimizing the code generation leading to even more efficient machine code [21].

6.3.1.2 Register Allocation

A simple and fast approach to register allocation is the Least Recently Used (LRU) register allocation scheme. As the name suggests, this involves always spilling the register that was least recently used whenever a value needs a register and one is not available.

Another scheme which is a little bit more involved is called the linear scan algorithm and involves liveness analysis. This algorithm calculates the span in which each variable is used and so can determine which variables are most appropriate to spill when new registers are required.

6.3.1.3 Constant Propagation

Figure 6.2 shows a simple example of constant propagation where lines one and two can be simplified to the expression on line four saving execution time. This type of constant propagation can be done entirely ahead of time but there are situations where compiling traces enable other types of constant propagation. Figure 6.3 shows an example of this where in a trace, `x` will be either two or four and as such the expression on line nine can be simplified. This simplification would not be possible ahead of time as the if statement would be compiled in its entirety.

```
1 int x = 4;
2 int z = 12 / x;
3 // Simplified by constant
    propagation to:
4 int z = 12 / 4;
```

Figure 6.2: Example of constant propagation.

```
1  int y;
2  ... // Some code
3  int x;
4  if(y < 10){
5    x = 2;
6  } else {
7    x = 4;
8  }
9  y = y - x;
10 ... // more code
```

Figure 6.3: Example of constant propagation enabled by tracing.

6.3.2 Trace Tree Optimization

The current approach to trace stitching is quite straight forward. However, constantly performing look-ups where to jump next can become costly in the long run. One solution to this problem is to instead modify previously compiled traces with direct jumps to side exit traces, reducing the overhead introduced by looking up jump destinations. This approach involves keeping track of where each instruction is located in memory which requires more control over the assembled traces than we currently have in our implementation.

Another approach is to at certain points take a trace together with its compiled side exit traces (this can be referred to as a trace tree) and recompile it to a single unit opening up for optimizing the separate traces together. This recompilation can be done on a separate thread and thus not block the execution of the rest of the program.

Implementing trace tree optimization would solve the problem with excessive memory consumption of some programs discussed in Section 5.2.3.

6.3.3 Showing Source Code in Visualizer

While showing byte-code instructions in the visualizer provides some understanding of the inner workings of the tracing JIT compiler, the connection to the source Java code is sometimes hard to see. Therefore, as an addition to the byte-code instructions, we think that it would provide even more understandability to show the source code and similarly color the its recorded parts. In the Java class file, there are some attributes that provide debug information to connect byte-code instructions to lines in the source code.

6.3.4 Scientific Evaluation of Understandability

In this thesis, we have not evaluated the success of understandability in any clear way. Instead, we have made a case for it by describing the steps taken which we believe contribute to the understandability. The actual judgment of success is therefore handed over to the reader as in its current form, understandability is a very subjective goal.

A more scientific study can be carried out where a group of developers get to familiarize themselves with our project and then get an assignment to, for example, implement a new feature or change some behavior of the compiler. We can then evaluate how easily they are able to complete this task compared to a similar task in a different tracing JIT compiler.

In this way, we can get a better outside opinion of understandability and since we try to measure how easily the project can be adopted, the actual degree of understandability can, in some sense, be measured.

6.4 Conclusion

Tracing JIT compilers can be useful tools when AOT compilation is not an option, either because short start-up times are crucial or because the dynamic nature of the programming language makes AOT compilation infeasible. Tracing JIT compilation involves several complicated processes that work together to form a complex system.

We have shown that by separating the functionality of these processes into distinct modules with clear areas of responsibility, one can get a clear overview of the processes and how they interact. We have implemented a tracing JIT compiler designed for understandability which we have evaluated on a small, but relevant, collection of examples. Additionally, we have implemented a run-time visualizer capable of exposing some of the inner functionality of our tracing JIT compiler that would otherwise be hidden from the user.

As far as we know, this is the first tracing JIT compiler designed with this objective accompanied by a run-time visualizer tool.

Bibliography

[1] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, vol. 35, no. 5, p. 1–12, 5 2000. [Online]. Available: www.hpl.hp.com/cambridge/projects/Dynamohttps://doi.org/10.1145/358438.349303https://doi.org/10.1145/349299.349303

[2] "SpiderMonkey Internals." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals

[3] "V8 Development Blog." [Online]. Available: https://v8.dev/blog

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, I. Rogers, O. Zendra, and E. Jul, "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*, ser. ICOOOLPS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 18–25. [Online]. Available: https://doi.org/10.1145/1565824.1565827

[5] G. Garen, "Announcing SquirrelFish," 2008. [Online]. Available: https://webkit.org/blog/189/announcing-squirrelfish/

[6] T. Kalibera, P. Maj, F. Morandat, and J. Vitek, "A Fast Abstract Syntax Tree Interpreter for R," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 89–102. [Online]. Available: https://doi.org/10.1145/2576195.2576205

[7] I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 291–300. [Online]. Available: https://doi.org/10.1145/277650.277743

[8] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V, vol. 27, no. 9. New York, NY, USA: Association for Computing Machinery, 1992, p. 85–95. [Online]. Available: https://doi.org/10.1145/143365.143493

[9] I. of Electrical, I. S. Electronics Engineers, J. R. Larus, T. Ball, and J. R. Larus, "Efficient Path Profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, 1996, p. 46–57. [Online]. Available: https://dl.acm.org/citation.cfm?id=243857

[10] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, CGO*, 3 2004, pp. 75–86. [Online]. Available: http://llvm.cs.uiuc.edu/

[11] A. Gal, C. W. Probst, and M. Franz, “HotpathVM: An Effective JIT Compiler for Resource-constrained Devices,” in *VEE 2006 - Proceedings of the Second International Conference on Virtual Execution Environments*, vol. 2006, 2006, pp. 144–153.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, “Trace-Based Just-in-Time Type Specialization for Dynamic Languages,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09, vol. 44, no. 6. New York, NY, USA: Association for Computing Machinery, 6 2009, p. 465–478. [Online]. Available: https://doi.org/10.1145/1543135.1542528https://doi.org/10.1145/1542476.1542528

[13] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, M. Franz, R. Rick, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz†, “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*, ser. VEE ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 71–80. [Online]. Available: https://doi.org/10.1145/1508293.1508304

[14] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, “The Java® Virtual Machine Specification,” Tech. Rep. 15, 2020. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se15/jvms15.pdf

[15] R. Isemann, “Beyond Debug Information : Improving Program Reconstruction in LLDB using C ++ Modules,” Master’s thesis, Chalmers Univesity of Technology, 2019. [Online]. Available: https://hdl.handle.net/20.500.12380/300037

[16] “Visual Studio debugger documentation.” [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/debugger/?view=vs-2019

[17] “GDB: The GNU Project Debugger.” [Online]. Available: https://www.gnu.org/software/gdb/

[18] “The LLDB Debugger.” [Online]. Available: https://lldb.llvm.org

[19] “AsmJit Documentation.” [Online]. Available: https://asmjit.com

[20] “OpenJDK.” [Online]. Available: https://openjdk.java.net

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, Oct. 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

A
Appendix 1

Raw LLDB output discussed in Chapter 4

```
Process 8733 resuming
General Purpose Registers:
       rax = 0x00007ffeefbfed08
       rbx = 0x0000000000000000
       rcx = 0x203a736574617453
       rdx = 0x0000000000000002
       rdi = 0x00007ffeefbfed08
       rsi = 0x00007ffeefbfe870
       rbp = 0x00007ffeefbfee80
       rsp = 0x00007ffeefbfeab0
        r8 = 0x0a31203a73657461
        r9 = 0x0000000000000000
       r10 = 0x0000000000000002
       r11 = 0x0000000000000499
       r12 = 0x0000000000000000
       r13 = 0x0000000000000000
       r14 = 0x0000000000000000
       r15 = 0x0000000000000000
       rip = 0x0000000100159797  TigerShrimp`RunTime::run(Program*) + 407 at RunTime.cpp:20:20
    rflags = 0x0000000000000246
        cs = 0x000000000000002b
        fs = 0x0000000000000000
        gs = 0x0000000000000000
(std::stack<Value, std::deque<Value, std::allocator<Value> > >) state->stack = {
  c = size=1 {
    [0] = {
      type = {
        type = Int
        subType = 0x0000000000000000
      }
      val = (intValue = 1, longValue = 1, floatValue = 1.40129846E-45, doubleValue = 4.9406564584124654E-324)
    }
  }
}
(std::map<unsigned long, Value, std::less<unsigned long>, std::allocator<std::pair<const unsigned long, Value> > >) state->locals =
      size=3 {
  [0] = {
    first = 1
    second = {
      type = {
        type = Int
        subType = 0x0000000000000000
      }
      val = (intValue = 1, longValue = 4294967297, floatValue = 1.40129846E-45, doubleValue = 2.121995791459338E-314)
    }
  }
}
(ProgramCounter) state->pc =
(size_t) method = 23
(size_t) pc = 5
(std::map<ProgramCounter, unsigned long, std::less<ProgramCounter>, std::allocator<std::pair<const ProgramCounter, unsigned long> >
      >) profiler.loopRecord = size=1 {
  [0] = (first =
(size_t) method = 23
(size_t) pc = 4
, second = 1)
}
(JVM::Mnemonic) $26 = (Mnemonic) mnemonic = SIPUSH
```

Figure A.1: Example of raw LLDB output.

B
Appendix 2

Code examples that were too large to fit in Chapter 5.

```java
public class EvenMoreLoops {
  public static void main(String[] args) {
    int a = 1;
    int i = 0, j = 0, k = 0, l = 0, m = 0, n = 0, o = 0, p = 0,
        q = 0, r = 0, s = 0, t = 0, u = 0, v = 0, w = 0;
    for (; i < 4; i++) {
      a += i;
    }
    for (; j < 4; j++) {
      a += i + j;
    }
    for (; k < 4; k++) {
      a += i + j + k;
    }
    for (; l < 4; l++) {
      a += i + j + k + l;
    }
    for (; m < 4; m++) {
      a += i + j + k + l + m;
    }
    for (; n < 4; n++) {
      a += i + j + k + l + m + n;
    }
    for (; o < 4; o++) {
      a += i + j + k + l + m + n + o;
    }
    for (; p < 4; p++) {
      a += i + j + k + l + m + n + o + p;
    }
    for (; q < 4; q++) {
      a += i + j + k + l + m + n + o + p + q;
    }
    for (; r < 4; r++) {
      a += i + j + k + l + m + n + o + p + q + r;
    }
    for (; s < 4; s++) {
      a += i + j + k + l + m + n + o + p + q + r + s;
    }
    for (; t < 4; t++) {
      a += i + j + k + l + m + n + o + p + q + r + s + t;
    }
    for (; u < 4; u++) {
      a += i + j + k + l + m + n + o + p + q + r + s + t + u;
    }
    for (; v < 4; v++) {
      a += i + j + k + l + m + n + o + p + q + r + s + t + u + v;
    }
    for (; w < 4; w++) {
      a += i + j + k + l + m + n + o + p + q + r + s + t + u+v+w;
    }
    System.out.println(a);
  }
}
```

Figure B.1: Test case: EvenMoreLoops.