Area 1 (Virtual Machines and Compilation)

Danny Yoo (dyoo@cs.wpi.edu)

Abstract

The ideas behind tracing JITs go back to the late 1970s, but have become especially popular lately. Explain the principles behind a tracing JIT (such as those in JavaScript and Lua). What assumptions do a typical modern tracing JIT make about its language, and what impact do various language control features such as function calls, tail-calls, exceptions, cooperative multitasking, preemptive multitasking, and continuations have on the tracing JIT's ability to improve performance?

1 Introduction

Dynamically-typed languages such as JavaScript and Lua dominate their respective domains. JavaScript is the language of the web, and Lua is the predominant embedded programming language for video games. Programs in these languages are often transferred in source-form across the network for other users to execute. As a consequence, the evaluation strategies for these languages place a high premium on portability and size. This scenario makes an interpretation strategy attractive for evaluation, especially compared to a more heavyweight compilation approach on the server side. Interpretation, however, can introduce some overhead. Furthermore, dynamic typing also introduces significant runtime overhead due to the cost of type dispatch. By compiling programs on the client side, a language implementation can eliminate interpretive overhead, but because the environments of Lua and JavaScript are interactive, compiling resources are limited. A dynamic compiler can't use too much time and space toward compilation or risk attracting notice from the end user.

JIT compilation techniques offer a balanced approach by focusing compilation efforts, not on the entire program, but only on promising regions. Tracing JITs, in particular, use runtime profiling information to focus their efforts on loops and previous iterations through a program, and can apply optimizations based on the observation of types at runtime. By taking advantage of type specialization, tracing JIT compilation allows dynamically-typed language evaluators to perform competitively with those of statically typed languages, with minimal cost to compilation resources.

2 Tracing JITs

A JIT can improve compilation performance by limiting compilation only to areas of code that it heuristically determines is worthwhile. Tracing JITs depend on a few assumptions:

1. The evaluation of loops dominates the runtime of a program.
2. Repeated runs follow the same control paths.
3. The primitive operations applied during evaluation can be accurately recorded at runtime.

When these assumptions hold true, then a tracing JIT can focus its compilation efforts on these loops and expect to gain from its efforts.

The third assumption, to monitor evaluation, is immediately true for interpreters, which is why tracing JITs and interpreters are a good fit for each other. However, interpretation is not a requirement for tracing JITs. A native-code evaluator, such as the one used in SPUR [1], which uses instrumented native code that communicates its operations, can be coupled to a tracing JIT.

2.1 Loop detection

Tracing JITs act by monitoring evaluation and identifying promising regions of code at runtime. Modern tracing JITs focus on loops since they are good candidates for hot spots in a program's evaluation. They heuristically find loops in two main ways:

- Dynamically: by watching the program counter (PC) for backwards jumps
- Statically: by annotating language features, such as high-level loop structure, within the source program

The dynamic approach uses knowledge about the program counter (PC): if the program counter goes backwards, it treats the backwards branch as a loop. Systems such as DynamoRIO [9] and PyPy [2] use this approach. A consequence of this low-level approach is its robustness: not only are loops directly encoded with a language's native iteration forms detectable, but loops encoded with tail calls and direct jumps, as in languages like Scheme, can also be detected by the program counter heuristic. Another side effect of using the PC is that it effectively applies a loop-unrolling in the presence of nested loops [7], which may improve performance at the cost of some code expansion.

The dynamic approach also exposes loops that are encoded by indirect, computed jumps. Examples of such kinds of loops are somewhat more unusual, but one example is due to a kind of cooperative multitasking involving consumers and producers. A producer/consumer coroutine can be constructed where indirect jumps are applied to pass control from one coroutine to the next, effectively constructing a loop between the two. In principle, when a set of coroutines stays fixed and the control flow follows predictably from one coroutine to the next, then dynamic PC approach can detect the loop and be traced.

In contrast, the static approach acts by instrumenting the program, before evaluation, at linguistic positions where loops are likely to occur. TraceMonkey's implementation, for example, dedicates efforts to loops that are encoded directly with the high-level "`for`" loop language construct. During bytecode compilation, these loops are assigned a dedicated loop header bytecode to triggers the tracer to start monitoring evaluation. In this kind of system, the runtime can apply self-modifying code techniques to toggle a loop's tracing by editing the loop header bytecode accordingly.

A positive benefit of this approach is that nested loops that are encoded directly in high-level language constructs are easy to detect, and a refinement to tracing JITs allow nests of loops to be handled efficiently and without duplicating compiled code [6]. However, one unfortunate consequence is that loops encoded by tail calls will not be detected by this heuristic alone, nor loops encoded with indirect jumps.

2.2 Trace anchors

The heads of these loops are known, more generically, as *trace anchors*. When evaluation passes repeatedly through these points past a certain threshold, the tracing JIT's runtime begins to record primitive operations to form a linear trace of the evaluation. If the loop completes its evaluation and control flows back to the beginning of the loop, then the recorded trace is serialized into native code, and subsequent runs through the loop use the compiled trace code. The trace structure, at this point, might not be immediately discarded, as subsequent runs through the compiled trace may motivate a revised look of its structure.

Some tracing JIT implementations generalize the notion of trace anchors to include non-loop positions as well as loops. By expanding static annotation from high-level loops to other positions in the language, then even tail calls can be be treated as loops. SPUR, for example, adds trace anchors on the heads of recursive functions. This extension allows SPUR to successfully detect loops encoded with iterative tail recursion at the cost of expanding the candidate set and increasing compile time, since more of the program may be considered and traced.

2.3 Tracing

A tracing JIT observes and constructs a running log of the primitive operations that are applied. It needs to record the operations at a fine-enough granularity so that it can construct a compiled native-code stream

that performs a computation equivalent to the observed one.

As an example, we can consider the following following code snippet in Lua.

```
function printMessage(n)
    write(n)
    write(" bottles of beer on the wall")
end

n = 99
while n > 1 do
    printMessage(n)
    n = n-1
end
```

When this executes, a tracing JIT can infer a trace anchor at the head of the `while` loop. On iteration through the loop, the anchor activates the tracing JIT's recorder to monitor the following stream of primitive operations:

```
trace-start:
    test that n > 1
    push the stack frame used for the procedure call to printMessage
    dynamic dispatch to write the number n (read from the newly constructed stack frame)
    dynamic dispatch to write the string " bottles of beer on the wall"
    pop the stack frame
    decrement n
    jump back to trace-start
```

An implication of tracing is that the operations of function calls during a trace are transparent to the recorder, effectively inlining function calls into a trace. In fact, some of the operations may be unnecessary due to the nature of the inlining. In the example above, the construction of the stack frame is redundant in principle because the value of `n` in the frame is the same as the one in the variable, and the stack frame pushing and popping can be treated as dead code. This knowledge allows a tracing JIT to optimize function calls that run in the dynamic context of a trace.

This example also introduces the notion of the need for some operations to perform type dispatch; `write` is a operation that works on both numbers as well as strings. Section 3 discusses the implications of type dispatch and tracing JITs in more detail.

2.4 Coping with exceptions and failures

Tracing may fail for simple reasons: the trace may ends up being too long, since tracing records the dynamic (potentially unbounded) behavior of a program, rather than the static representation of a program. Tracing may also fail, however, for more subtle reasons.

1. It may encounter a primitive operation that is unobservable.
2. It may encounter a primitive operation whose behavior is dynamic and complex.

A failure of the first type is due to a violation of the third assumption of tracing: the core evaluator performs an operation that the tracer can't record with the required granularity. An example of an unobservable operation would be the access of the DOM from TraceMonkey. TraceMonkey's trace recorder works on the level of JavaScript bytecode, and its recorder doesn't have the ability to monitor the internals of the C++ implementation of DOM operators.

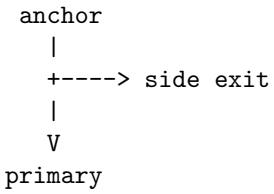
Examples of the second failure type include the throwing of an exception or the application of a continuation. Each of these is an operation that has substantial dynamic behavior that changes the state of

the control context; trying to record the primitive operations performed is probably a waste of compilation effort, because the next occurrence of the exceptional condition will have a different dynamic effect on the control context.

In the face of a tracing failure, the behavior of a tracing JIT varies. Some implementations, such as TraceMonkey, will immediately abort the trace, throwing away the recorded trace up to that point. Other implementations will heuristically decide to retain the trace up to the failure, ending with code that allows the existing evaluator to pick up evaluation where the trace left off. These *side exits* are described next.

2.5 Side exits and extending traces to trees

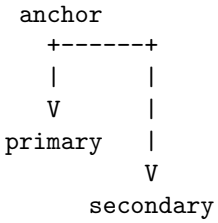
The second assumption of JITs, that repeated evaluations of a loop follow the same control flow, is obviously not always true: otherwise, that would imply that all programs are infinite loops. During subsequent runs through a compiled trace, if control flow takes an alternative path to the one that was recorded, then the compiled code will take a *side exit*. Such side exits are placed at branch points in the original program, and represent paths that haven't been recorded yet. When a side exit is taken, the code at the side exit reconstructs the state necessary to restart the original evaluator, and continues computation from where the compiled trace left off.



This reconstruction can be expensive, since it may involve synthesis of stack frames and boxing of values. For example, if the trace's execution is in the middle of evaluating a function call when it hits a side exit, the associated side exit code must construct the associated stack frame before transferring control to the original evaluator to resume the computation. In essence, hitting a side exit forces the synthesis of the computation's continuation.

Therefore, a tracing JIT will attempt to have program control flow stay within natively-compiled code as much as possible. One strategy used in modern tracing JITs is to extend traces beyond simple linear structures to *trace trees*. Tracing JITs will account for minor violations of the second assumption by allowing primary traces to be extended and account for branches within the dynamic context of a loop.

When evaluation through a compiled trace flows into a side exit, while the tracing JIT reconstructs the state necessary to restart the evaluator, it also begins to record this alternative control-flow path in an attempt to construct a secondary trace for the loop. If this alternative control flow manages to flow back to the beginning of the loop, then the tracing JIT patches this secondary trace to branch off of the primary trace, replacing the existing side exit. It then compiles each of the observed traces to native code. The structure of the trace is a *trace tree* consisting of the primary trace and its branches.



2.6 Preemptive multitasking, dispatch loops, and interpreters

The tracing JIT within TraceMonkey enables preemptive multitasking by adding a preemption guard to the head of each compiled trace: the compiled trace will check to see if a preemption flag has been set, and if so, immediately follow a side exit to relinquish control to the scheduler.

Although it would be technically possible to apply tracing itself to the toplevel event loop that implements the preemptive multitasking engine, doing so would most likely be ineffective because of the inherent branchiness and unpredictability of the dispatch loop. The unpredictability causes excessive side exit transfers, which introduce overhead.

The unpredictability of dispatch loops can greatly affect a tracing JIT for a particular class of programs: interpreters. The performance of a language interpreter program that runs in a basic, naive tracing JIT can be poor, not because the tracer isn't tracing, but because the tracer focuses on the interpreter opcode-dispatch loop, which results in traces that are too short to provide a performance benefit. But unlike event dispatching, at a higher level, the control flow is following predictably through the user's code, if the program flow is measured as the one represented virtually by the interpreter rather than by the native program counter.

An extension to a tracing JIT can allow the tracing mechanism to trace through the interpreter into an interpreted program. If the tracing JIT itself provides hooks for the language implementor to provide runtime hints to expose logical loop structure, to virtualize the program counter, then the tracing JIT can effectively construct more effective loop traces that are expressed in terms of loops within the user-level program [9] [2]. Exposing these hints gives the implementor an easy path to integrate a tracing JIT effectively into an interpreter.

2.7 Compilation and general optimization opportunities

The use of the tracing JIT compiler allows the runtime to generate native code by compiling traces. A trace is recorded in static single assignment (SSA) form, which provides opportunities for aggressive optimizations to be performed on during a trace's compilation [4] [5]. Since an individual trace is linear and in SSA, the tasks of determining variable liveness, propagating constants, and eliminating dead code are all fairly immediate for the optimizing compiler.

The use of these optimizations has an impact on the performance of function calls within a trace. In the context of a trace, when a function is called, the operations of that function call are also recorded during tracing, its primitive operations collected as a part of the trace. The optimizations enabled by SSA effectively perform intraprocedural optimizations, eliminating the use of the data and control stack operations that would be otherwise required for a general procedure call, effectively performing function inlining.

3 Applying tracing JITs to dynamically-typed languages

Tracing JITs provide benefits regardless if the compiled language is statically or dynamically typed: they provide lightweight dynamic compilation that can be retrofitted on top of an existing evaluation strategy. Preliminary work on dynamic compilation on trace trees, for example, applied a a tracing JIT to statically-typed Java [7].

For dynamically typed languages, though, tracing JITs offer another opportunity for optimization with regards to dynamic types. In a dynamically-typed language, types are associated to values rather than to names or locations. In many dynamic language implementations, even primitive values such as bytes and integers need to carry along type tags in order to properly distinguish between the types during runtime [8]. These languages also provide operators that work across different types. In Python, for example, the addition operator works not only on numbers, but also on strings, booleans, lists, and on anything that implements Python's `__add__` protocol. A dynamically-typed language's operators naively needs to perform a type-dispatch to choose which appropriate action to perform, and this conditional check contributes to the lack of performance.

We revisit the second assumption made by tracing JITs: within the running of loops, a tracing JIT assumes that it's likely that the same kind of control flow decisions will be made. In a dynamic language, many of these control flow decisions are due to the type dispatch of dynamic operations. A compiled trace can be specialized for the types that are observed during runtime [3]. A tracing JIT can extend its assumptions, and speculate that the same type-dispatching decisions are executed within a loop. If it's the case that the same type-dispatching decisions are made between loop iterations, then this permits the JIT compiler to eliminate the type dispatch code.

As an example of why tracing JITs are especially effective for dynamic programs, we can consider a JavaScript program that computes the Gauss sum of the first 100 integers.

```
var i, sum = 0;
for (i = 0; i < 100; i++) {
    sum += i;
}
```

Due to JavaScript liberal semantics, even a simple program such as the Gauss sum above extensively uses several dynamic operators, including `<`, `+`, and `++`. The type dispatching is based entirely on the types of the operands. `<` may be either lexicographic or numeric comparison, depending on the types of its operands, and likewise, the `+` operator in JavaScript may do string concatenation as well as arithmetic addition. If the input to `++` is a string, JavaScript will automatically force a coercion to a number before incrementing it.

A portion of a run through this program will look something like this:

```
1.  sum = 0
2.  i = 0

3.  dynamic dispatch for '<' based on the types of i and 100:
4.      -> test the result of the numeric comparison of i and 100.
5.  dynamic dispatch for '+' based on the types of sum and i:
6.      -> sum = the result of the numeric addition of sum and i.
7.  dynamic dispatch for '++' based on the type of i:
8.      -> increment i.

9.   dynamic dispatch for '<' based on the types of i and 100:
10.      -> test the result of numeric comparison of i and 100.
11.  dynamic dispatch for '+' based on the types of sum and i:
12.      -> sum = the result of the numeric addition of sum and i.
13.  dynamic dispatch for '++' based on the type of i:
14.      -> increment i.

...
```

The statements from 3 to 8 represent a trace through the loop, and the statements from 9 through 14 are another iteration of the loop. The JIT trace compiler can observe a loop between 3–8, and see that within that run that the types of i and sum are numeric.

Each dynamic dispatch is expensive, and any optimizations that can eliminate them are worthwhile. Therefore, the trace compiler speculates that the input types on each iteration of the loop are, most likely, stable. Tracing JITs make one more assumption with regards to primitive operations, that it can predict what the result types of operands are. Assuming that all the results of primitive operations have predictable types, then the compiler can produce code that specializes for numeric types:

```
assuming sum and i are numeric:
    test the result of the numeric comparison of i and 100.
    sum = the result of the numeric addition of sum and i.
    increment i.
```

where the head of the code first checks that it's assumptions on the initial types of the variables matches that of the observed, recorded run. The generic dynamic dispatch operations, being redundant, can be eliminated, leaving behind only the type-specialized operations to be compiled.

Knowing that these are primitive values also allows the JIT compiler to choose a cheaper, unboxed representation within the context of the trace. Rather than treat all numbers as floating point, a tracing JIT may use unboxed integers representations.

```
assuming sum and i are integers:
    test the result of the integer comparison of i and 100.
    sum = the result of the integer addition of sum and i.
    integer-increment i.
```

Once the compiler exits a trace, the values can be coerced back into the representations native to the core evaluator.

This particular optimization, though, presents a complication. The assumption that primitive operations produce values of predictable types is what motivates a tracing JIT to remove redundant type checks. However, some operations may not produce predictable output types. When numbers are represented as integers, for example, operations like addition may overflow, breaking the assumption. In this case, the tracing JIT compiler will generate the type-specialized code, but also introduce tests after such operations to verify that the values are of the expected types, the violation of which will lead to a side exit.

```
assuming sum and i are integers:
    test the result of the integer comparison of i and 100.
    test for integer overflow
    sum = the result of the integer addition of sum and i.
    test for integer overflow
    integer-increment i.
    test for integer overflow
```

If overflow occurs, as with other cases where control flow follows a different path than the one in the trace, the trace takes its side exit to finish the rest of the computation. However, this most likely leads to another trace of the loop with different assumptions on the input type, leading back to native-code evaluation with a compiled trace.

4 Summary

To revisit the questions:

- A modern tracing JIT assumes that: (1) loops dominate the runtime behavior of a program, (2) control flow follows predictable paths, and (3) evaluation is observable. In the context of dynamically-typed languages: (4) type-specialized operations return values of predictable types.

  Major violations of these assumptions leads to poor performance for a tracing JIT, because it causes evaluation to transfer out of the optimized, compiled code back to the original evaluator.

- Function calls, tail calls, and some forms of cooperative multitasking within a trace's context pose no fundamental difficulties to a tracing JIT, as their evaluation involves predictable control flow paths. In fact, due to the inlining nature of a trace, the abstraction cost of the general procedure calling convention can be eliminated when a tracing JIT produces the optimized compiled trace.

- Preemptive multitasking can be approached by introducing a side exit at the head of a trace anchor that checks a flag for a preemption request. Taking a side exit, in essence, forces the explicit construction of the continuation.

- Features like exceptions and continuations pose difficulties to tracing JITs because their effects have a dynamic impact on control flow, which violates one of the assumptions made by tracing JITs. Tracing JITs abandon attempts to compile these dynamic operations, and transfer control back to the original evaluator.

References

[1] Michael Bebenita, Florian Brandener, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. Technical Report MSR-TR-2010-27, Microsoft Research, 2010.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object Oriented Languages and Programming Systems*, 2009.

[3] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. Efficient Just-In-Time Execution of Dynamically Typed Languages Via Code Specialization Using Precise Runtime Type Inference. Technical Report 07–10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2006.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

[5] Andreas Gal, Michael Bebenita, Mason Chang, and Michael Franz. Making the Compilation "Pipeline" Explicit: Dynamic Compilation Using Trace Tree Serialization. Technical Report 07-12, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

[6] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[7] Andreas Gal and Michael Franz. Incremental Dynamic Code Generation with Trace Trees. Technical Report 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, 2006.

[8] David Guderman. Representing Type Information in Dynamically Typed Languages. Technical Report 93-27, The University of Arizona, 1993.

[9] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic Native Optimization of Interpreters. *Proceedings of the 2003 workshop on Interpreters, virtual machines, and emulators*, 2003.