Traditional JIT (just-in-time) compilers are method-based: they compile "hot" (i.e. frequently executed) methods to native code. An alternative is trace-based or tracing JITs, where the compilation unit is a (hot) sequence of instructions. Typically, such sequences of instructions correspond to loops, where programs spend most of their execution time.

Where did the idea of tracing come from? What was appealing about it? How was tracing adapted for JITs and dynamic languages? What happened to Mozilla's TraceMonkey, which used to be part of Firefox? Do any JITs today use tracing?

This document is based on my lecture notes for the *History of Programming Languages* course[1], as well as the discussion summaries recorded by Ben Greenman and Gabriel Scherer. The most recent version of this document can be found in the course repository[2].

In the presentation, I covered five papers, whose dependencies are summarized in Figure 1:

1. Bala, Duesterwald, and Banerjia. *Dynamo: A Transparent Dynamic Optimization System.* PLDI 2000. [2]

   The first tracing paper was published in 2000 by Bala et al. Dynamo was not a just-in-time (JIT) compiler, but was certainly influenced by JITs, along with dynamic translation and profile-guided optimization.

2. Gal and Franz. *Incremental Dynamic Compilation with Trace Trees.* UC Irvine Technical Report 2006. [10]

   In this technical report, Gal and Franz implemented tracing in a JIT compiler for Java. They also introduced the *trace tree* intermediate representation.

3. Gal et al. *Trace-based Just-in-Time Type Specialization for Dynamic Languages.* PLDI 2009. [12]

   In 2009, Gal et al. implemented TraceMonkey, a tracing JIT for JavaScript. TraceMonkey shipped as part of Firefox for several years.

4. Bolz, Cuni, Fijałkowski, and Rigo. *Tracing the Meta-Level: PyPy's Tracing JIT Compiler.* ICOOOLPS 2009. [7]

   Also in 2009, Bolz et al. implemented "meta-tracing" for PyPy [5], where they trace an interpreter instead of the user program. Their main inspiration was DynamoRIO [22], a project that adapted Dynamo for interpreters.

5. Ardö, Bolz, and Fijałkowski. *Loop-Aware Optimizations in PyPy's Tracing JIT.* DLS 2012. [1]

   In 2012, Ardö et al. adapted an optimization technique from LuaJIT [16], a tracing JIT for Lua, and implemented it in PyPy.

[1]`http://www.ccs.neu.edu/home/matthias/7480-s17/`
[2]`https://github.com/nuprl/hopl-s2017/tree/master/tracing-jit`

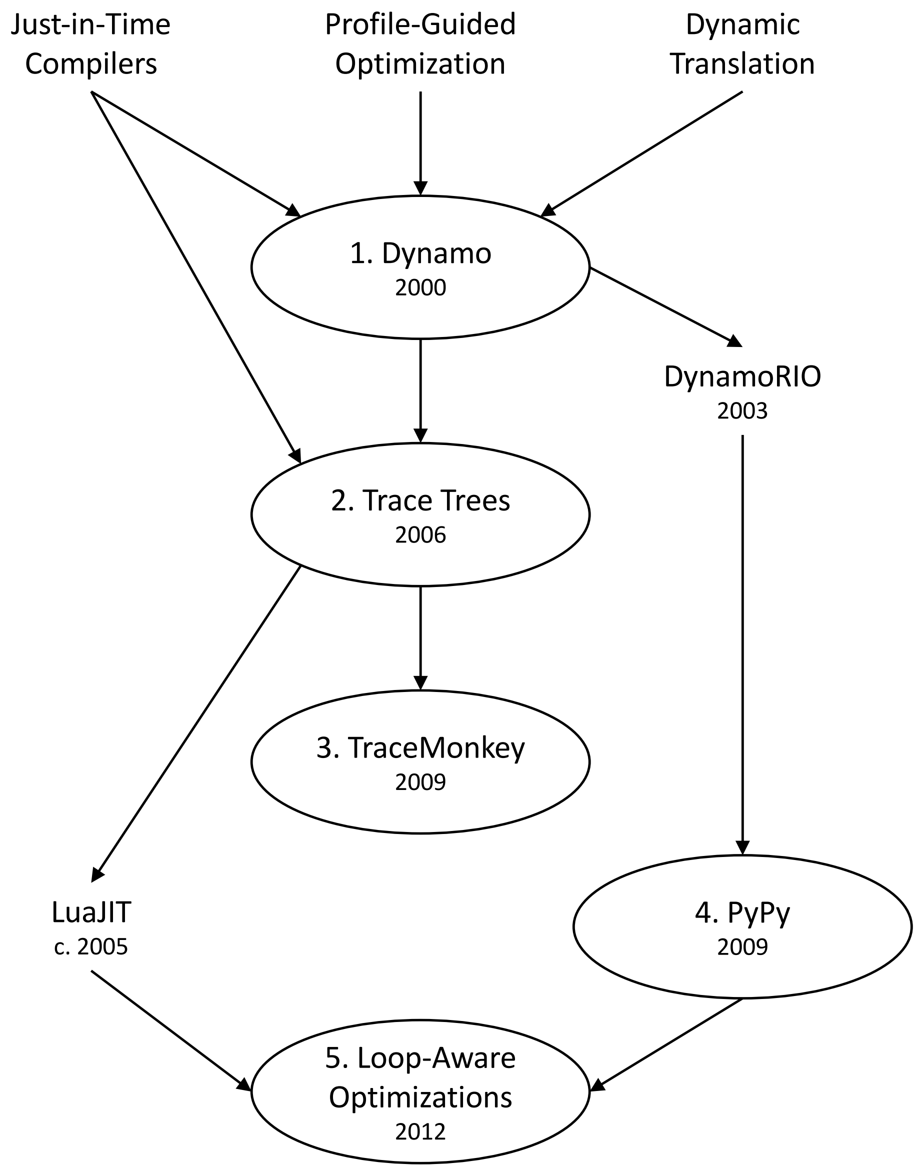

Figure 1: Dependencies of the papers discussed in these notes.

1 Dynamo

Dynamo: A Transparent Dynamic Optimization System
Bala, Duesterwald, and Banerjia. PLDI 2000. [2]

"Premature optimization is the root of all evil."
— Donald Knuth [13, p. 268]

This line is often quoted, but it is important to remember the context: a programmer should not waste time optimizing the "small inefficiencies." Instead, a good programmer will carefully identify the critical portions of the program and spend the optimization effort there. This is exactly what Dynamo does:

1. Interpret and monitor machine code to identify hot *traces*, i.e. critical portions of the program that are most likely loop bodies.

   X: Does the programmer need to write annotations?

   MHY: No.

   X: So if I have a loop and call out to another function—things you can do with λ spaghetti code . . . ?

   MHY: Yes. Write a program and compile it normally, and just give it to Dynamo. You don't need to do anything special.

2. Record the trace's sequence of instructions. A trace is identified by the address of its first instruction.
3. Optimize the trace and generate a *fragment* of more efficient machine code. Store the fragment in a cache (or array of bytes) so it can be executed later.

   X: What exactly is a fragment?

   MHY: More efficient machine code, fewer jumps.

4. During interpretation of the machine code, if a known trace is reached, execute the corresponding fragment on hardware.

Let us walk through Figure 2 as an example, while providing some more detail. The sample program contains branching, a function call and return, and a loop. Suppose the sequence of instructions ACDGHJE is executed before looping. When the program is laid out in memory and executed, control must jump over the B and I instructions. Furthermore, the two functions may not be near each other.

When Dynamo executes this program, its first step is to identify hot traces. Dynamo uses the *Most Recently Executed Tail* heuristic, which says that if an instruction is hot, it is very likely that the following sequence of instructions is also hot. Dynamo maintains a counter at *start-of-trace* points, which are targets of backward branches (likely to be loop

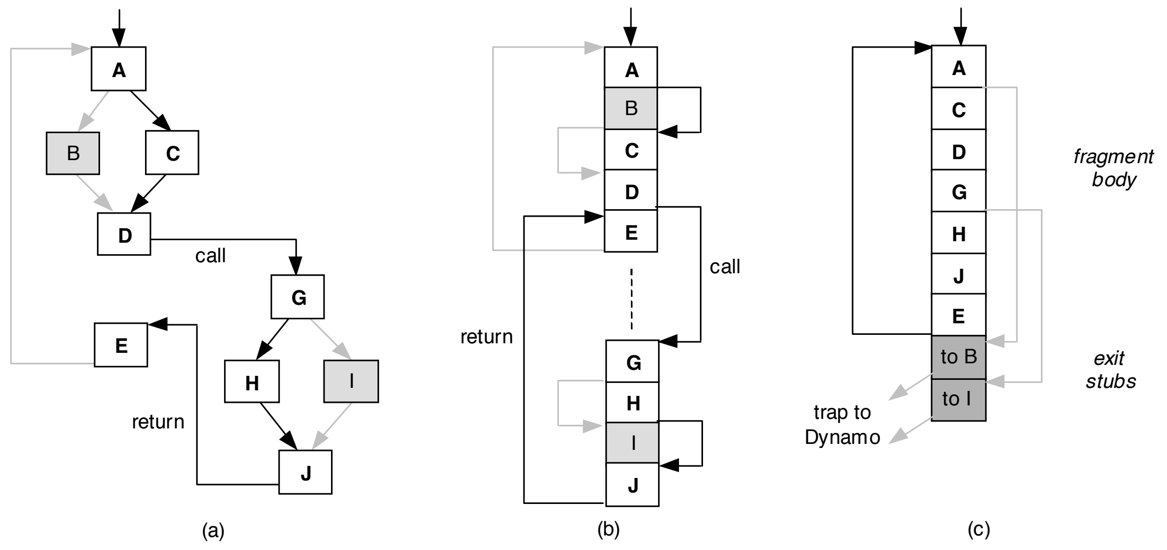

Figure 2: An example program. (a) depicts a control flow graph of two functions, (b) depicts a possible layout of the two functions, and (c) depicts a trace of the functions. [2, Fig. 3]

headers) and targets of trace exits. If the counter exceeds a certain threshold, Dynamo will record instructions until reaching an *end-of-trace* instruction, which is a backward branch or a target that corresponds to a known trace.

Once a trace is recorded, Dynamo transforms and optimizes it. Unconditional direct branches are removed, functions are inlined, and branches are rewritten so that the fast path is on the trace. The result is a single-entry, multi-exit sequence of instructions with no internal merge points. Figure 2 (c) shows a rewritten trace. The entire fast path is in a linear sequence, but branches to the slow path go to the exit stubs, and then return control to the Dynamo interpreter. The optimized trace is recompiled as a fragment, i.e. more efficient machine code, and stored in a code cache.

Whenever the interpreter executes a branch instruction, it checks if a fragment corresponding to the branch target exists in the cache. If so, Dynamo will context switch and execute the fragment on hardware. When the fragment finishes executing, control returns to the interpreter. Dynamo will always try to patch new fragments to jump directly to existing fragments, to avoid expensive context switches.

Why are traces so nice? Traces are linear sequences of code that can cross function boundaries, so they improve cache locality. Branches are rewritten to minimize the branch misprediction penalty. Furthermore, a trace is much simpler and faster to optimize, since there are no internal control flow merge points. Typical optimizations include redundancy removal, copy propagation, and constant propagation.

To test their implementation, the authors ran Dynamo on the SPEC CPU1995 integer benchmarks [21] as well as an incremental constraint solver called `deltablue` [20]. They

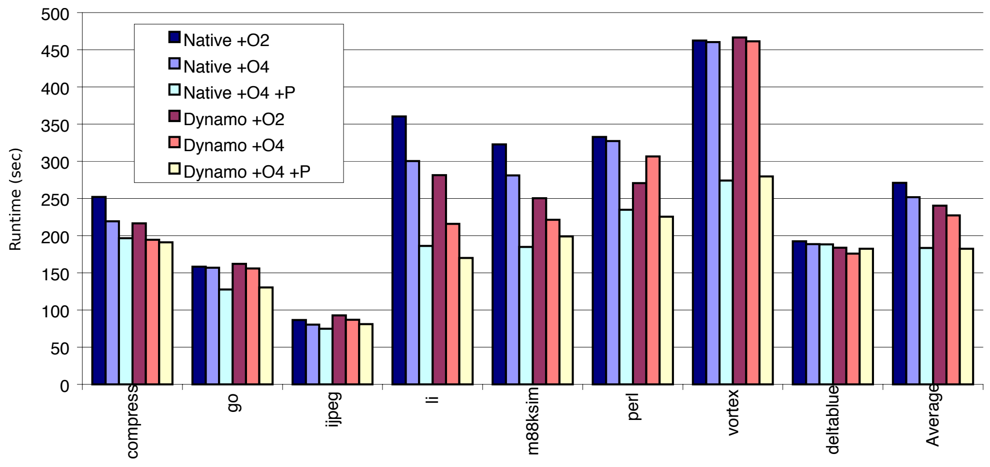

Figure 3: Dynamo performance on native binaries. The first three bars correspond to native executions, while the next three correspond to code run with Dynamo. [2, Fig. 8]

compiled the benchmarks with different levels of optimization (`+O2` for the first level, `+O4` for a higher level, and `+O4 +P` for a higher level with profile-guided optimization), and ran with and without Dynamo. The results are presented in Figure 3. The main takeaway is that `+O2` with Dynamo was slightly better than `+O4` without Dynamo.

X: The differences are not very big. Was this before the paper [9] that showed `-O3` optimizations were "within the noise"?

Y: First, this is a bad graph. The y-axis is in seconds. Nowadays, you would see normalized graphs.

X: But still, the difference is not that big.

Y: It's about 20%. The noise papers were more like 3%.

X: Did they talk about how Dynamo is no better than the state of the art? Native+PGO is better than Dynamo+PGO.

Y: Nobody was using PGO in practice.

MHY: The paper argued that nobody used +O4 either, because it was too slow and you lost all your debugging symbols.

Y: Does their system maintain debugging info?

MHY: OK, that wasn't the best example.

X: Did anyone ever look at tracing on the λ level, in terms of the number of β-reductions? Could see if this is generally a good idea.

MHY: No, not to my knowledge.

```
1: code;
2: do {
     if (condition) {
3:     code;
     } else {
4:     code;
     }
5: } while (condition);
6: code;
```

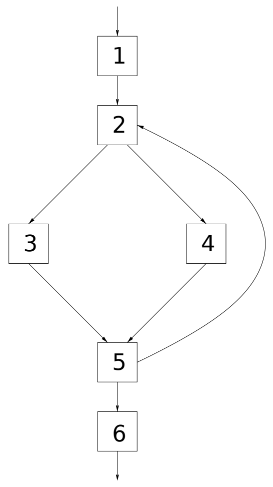

Figure 4: A sample program and control flow graph. [10, Fig. 1]

2 Trace Trees

Incremental Dynamic Code Generation with Trace Trees
Gal and Franz. UC Irvine TR 2006. [10]

We now move from tracing to tracing JITs, specifically a tracing JIT for Java. This technical report was never published; however, an earlier version of this work was published at a workshop [11].

> X: Did Gal go to Mozilla?
>
> MHY: Yes, he became CTO but left to do a startup.

Traditionally, JITs are method-based. They construct a control flow graph of the entire program, (e.g. Figure 4) and then generate native code for methods. In this work, the authors build a JIT around *tracing*, which can be done on-the-fly. This requires significantly less overhead, so such a JIT can be used on embedded devices.

The basic technique is similar to Dynamo, but the input is JVM bytecode instead of machine code:

1. Identify anchor nodes (e.g. targets of backward branches) and count the number of times they are hit.
2. If the hit count of an anchor node exceeds a certain threshold, start recording instructions. Stop recording when a cycle is found.
3. Compile the recorded trace to native code.

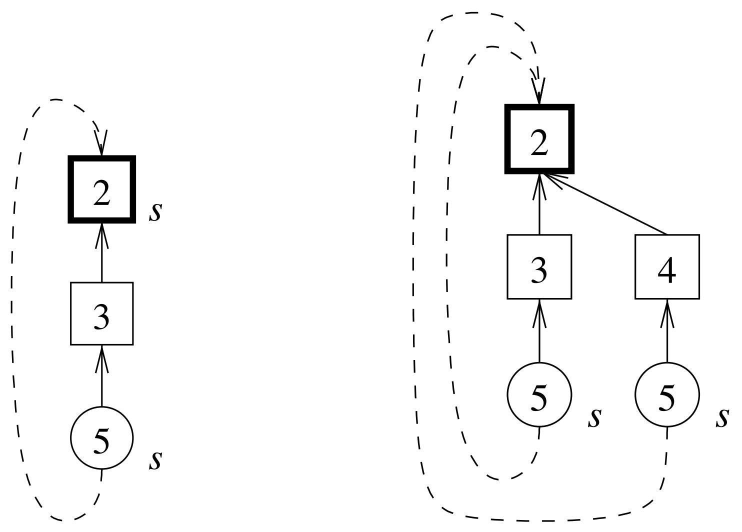

Figure 5: Two trace trees that correspond to the program in Figure 4. *s* represents the sites of possible side exits. The tree on the right is an extension of the tree on the right, formed when execution from block 2 goes to block 4 instead of block 3. [10, Fig. 3]

4. If known native code exists for some trace, execute it instead of interpreting the bytecode.

   X: Wouldn't OS developers complain about this behaviour?

   Y: This was 2006.

   Z: Racket does these things today. Let the OS developers complain.

5. Extend existing trace trees with new branches.

The last step is slightly different from Dynamo. In Dynamo, if a fragment was executing on hardware and a branch caused a side exit, the system starts recording a new trace and then compiles a new fragment. This might result in a tree-like structure, but Gal and Franz explicitly represent this structure as a trace tree. Note that traces within a tree can share a common prefix, but must always duplicate suffixes. This shared code structure can then be used to improve code generation.

As an example, consider tracing the execution of the program in Figure 4. Since block 2 is the header of a loop, it becomes the anchor node. Suppose we execute the following sequence of blocks: `235`. The resulting trace is displayed in Figure 5, left. Now, suppose the other path is taken within the loop, so the trace is `245`. During execution, the side exit at block 2 is taken, and then the trace `45` is recorded as a new branch, extending the current tree. The result is Figure 5, right.

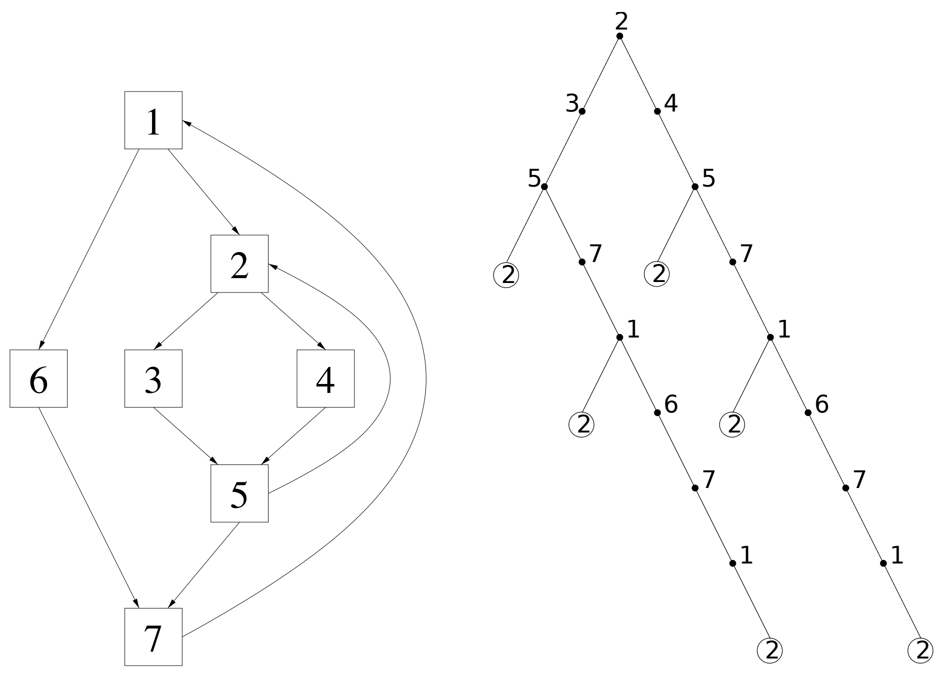

Figure 6: A nested loop and trace tree. Block 2 is selected as the anchor. Note the duplication of the outer loop in the trace tree [10, Fig. 4]

> X: What about program size?
>
> Y: It could be exponential. Do they do count duplicates in a tree? Do they bound the maximum width of the tree?
>
> MHY: No, they only keep track of the height of the tree. If the tree is too tall, they throw the trace away.

This becomes a significant issue with nested loops. Usually, the inner loop's header becomes the anchor, since it is hit more than the outer loop's header. When recording the trace, the loop is turned inside out, and the outer loop becomes part of the inner loop. This simplifies the loop structure at the expense of code duplication.

Figure 6 shows how a nested loop might be recorded. In this example, the inner loop header, block 2, is selected as the anchor. At first, only the inner loop is recorded: `235`. On the next iteration, control goes around the outer loop twice, before returning to the inner loop header: `23571671`. But later, a similar path is taken, but with the other branch in the inner loop: `24571671`. In this case, the outer loop is duplicated in the trace tree. This issue is not solved until we discuss the next paper.

The authors implemented a prototype in JamVM [14], a Java JIT for embedded devices, and ran the Java Grande benchmark suite [15]. Their performance is not quite as good as HotSpot, but they achieve this with significantly less effort and significantly less overhead.

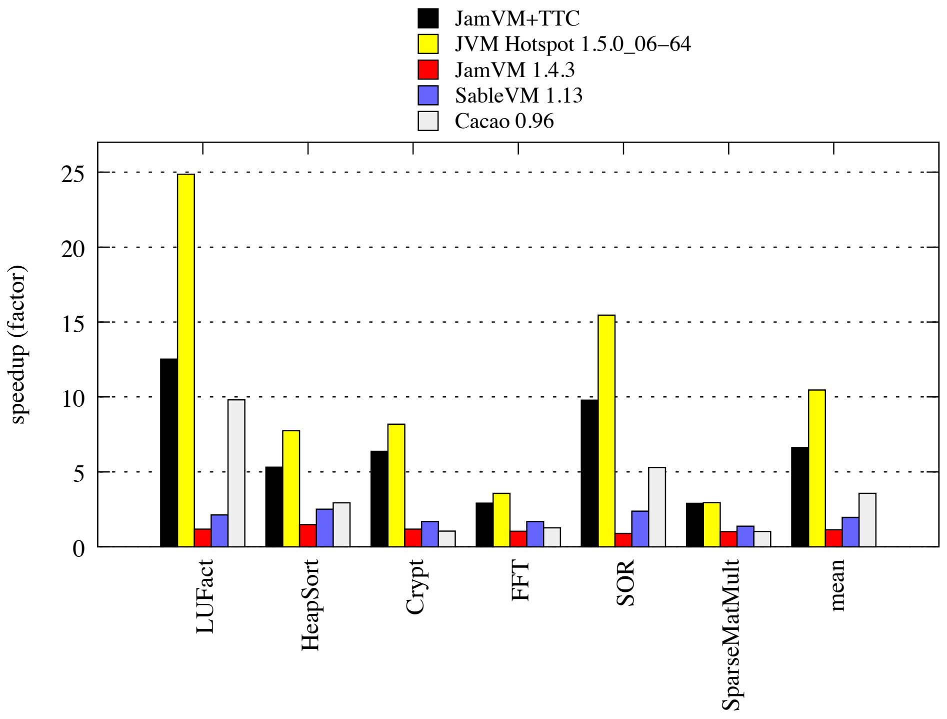

Figure 7: Performance of code compiled with different JVM implementations. The baseline is HotSpot using pure interpretation. [10, Fig. 8]

Specifically, using pure interpretation as the baseline, HotSpot achieves a 10x speedup while their tracing JIT is only 7x faster (Figure 7). However, their code generation is 350x faster than HotSpot (Figure 8 top), they generate 30x less code (Figure 8 middle), and they use 7x less memory (Figure 8 bottom). The takeaway is that it is much easier to construct and maintain a trace tree than a control flow graph.

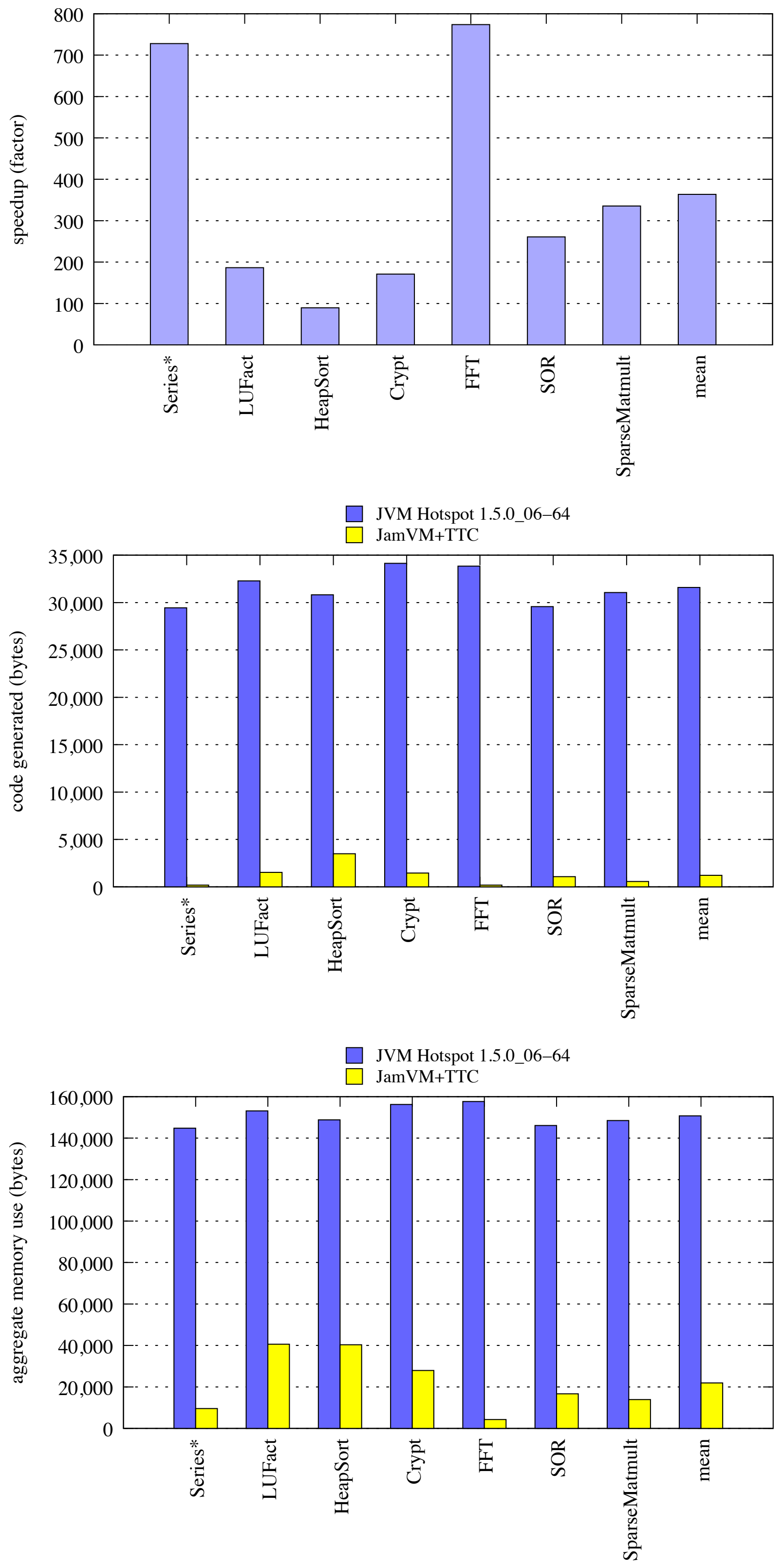

Figure 8: Compilation time [10, Fig. 9], generated code size [10, Fig. 10], and memory usage. The comparison is between HotSpot and their JamVM prototype. [10, Fig. 11]

3 TraceMonkey

Trace-based Just-in-Time Type Specialization for Dynamic Languages
Gal et al. PLDI 2009. [12]

In 2009, Gal, Franz, and many other authors adapt their tracing approach for a dynamic language, JavaScript. Eich, the creator of JavaScript, was a co-author.

> X: What do you mean by dynamic? Java has reflection and objects created at run time, and it's common in real world code. How is JavaScript different?
>
> MHY: In this context, "dynamic" refers to the lack of static type information. The previous work did not discuss reflection in Java.
>
> Y: Everyone knows reflection is hard to handle. See Wand's *The Theory of Fexprs is Trivial* [23]. It would be interesting to know the minimal conditions for a trivial theory.
>
> Z: The Fexprs paper just uses syntactic equality. The "theory" is contextual equivalence. Given terms M_1 and M_2, one context that distinguishes them is a context that diverges if its argument is α-equivalent to M_1. So contextual equivalence is trivial because it's only α-equivalence.

The assumption is that hot loops (traces) are mostly type stable, so the compiler can generate code that is specialized to the types. Each compiled trace corresponds to a single path, with specific variables mapped to types. They use the same trace tree approach as before, but now each anchor node has a type map. In other words, a trace is identified by the address of its first instruction as well as the type map. In a sense, the type map acts as a "function signature" of a trace.

> X: What do types mean here?
>
> MHY: Things like strings and numbers that can be unboxed and then optimized. JavaScript doesn't actually have integers; they're boxed and represented as floats. If you can unbox them and use actual integers, that's a huge performance win.
>
> Y: What's the type of the type map?
>
> MHY: Variables to types.

Again, extending traces is similar to before. However, the main difference is what is considered a cycle. In TraceMonkey, a loop is only "closed" if execution returns to the loop header *and* the types of the variables match the type map upon entry. If this is true, the loop is considered type-stable. Otherwise, the loop is type-unstable: it is treated as a single iteration with a side exit. A new trace is recorded with the new types, and the hope is that eventually, a type-stable loop is achieved. (In many cases, a loop will initially

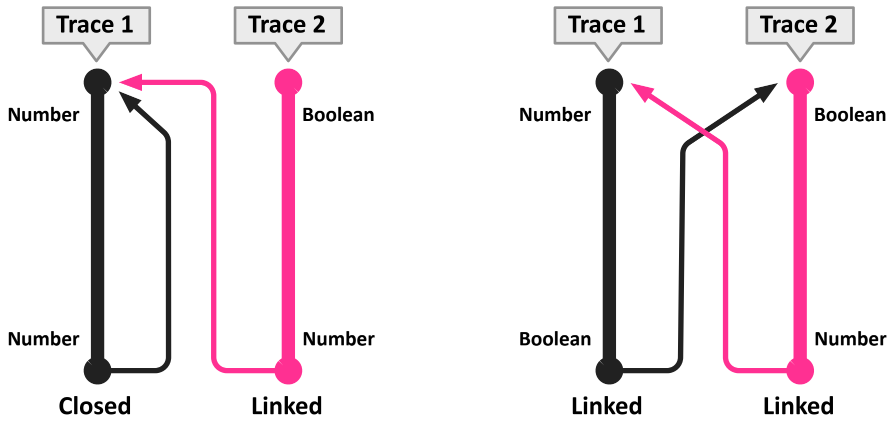

Figure 9: Linking two traces. On the left, Trace 1 is a closed loop, while Trace 2 cannot be closed, but can be linked to the beginning of Trace 1. On the right, neither loop is closed, but both can be linked to the other. [12, Fig. 6]

be type-unstable because some variables are uninitialized. Then, in subsequent iterations, variables are initialized and the loop becomes type-stable.)

X: In practice, this happens all the time in generic code.

Y: You mean, ad-hoc polymorphic code.

Z: This is on-the-fly monomorphization. It's just abstract interpretation [8]. In honour of my friend Patrick [Cousot], I'm going to say this twice a week for the rest of the semester.

X: Was this in SpiderMonkey? Which of the three compilation tiers was it in?

MHY: Yes, it was in SpiderMonkey, but I don't know which tier.

TraceMonkey will also try to link traces together to avoid side exits. Figure 9 shows an example of linking. On the left, Trace 1 is a type-stable loop, since the variables have type `Number` at the beginning and at the end of the iteration. Trace 2 is type-unstable, because the variables start with type `Boolean`, but finish as type `Number`. However, Trace 2 can then be linked to Trace 1. On the right, neither trace is type-stable, but both can be linked to the other.

If we recall the previous paper, they turned nested loops inside out. This simplified the implementation, since a nested loop is treated as a simple loop. However, the outer loop often gets duplicated. If we assume that loops are nested at a depth of k, and that each loop has n paths, then the number of traces will be $O(n^k)$.

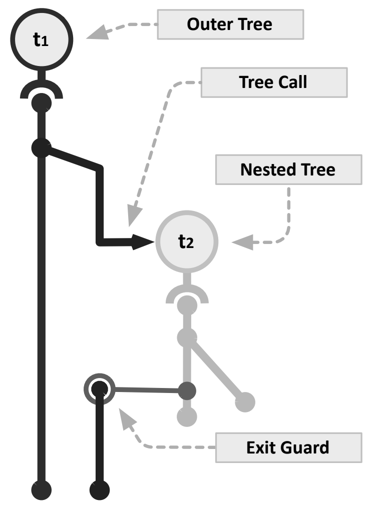

Figure 10: Nested trace trees. t_1 is the outer tree (L_R in the algorithm) and calls t_2, the inner tree (L_O in the algorithm). [12, Fig. 7]

In TraceMonkey, each loop is represented by its own trace tree. The inner and outer loops are treated as separate traces, and TraceMonkey keeps track of which is the inner loop and which is the outer loop. Then, TraceMonkey simply makes the outer loop call the inner loop, as if it were a function. This is described in the following algorithm:

- Start tracing from loop L_R's header.
- If there is a loop exit, stop the trace.
- If a different loop header L_O is reached:
  - L_O must be an inner loop of L_R, since we never exited L_R.
  - If L_O has a type-matching compiled trace tree, call the trace tree. If the call succeeds, record it in the trace so future executions can call the inner trace.
  - Otherwise, abort the trace of L_R and start recording L_O. In the next iteration, L_R will be recorded, and can then call L_O's compiled trace.

Figure 10 is an example of a nested loop, where the outer loop t_1 calls the inner loop t_2. If we assume that loops are nested at a depth of k and each loop has m different type maps, then the number of traces will be $O(m^k)$. Since loops are mostly type-stable, we assume $m \approx 1$.

To evaluate TraceMonkey, the authors ran the SunSpider benchmark suite [24] and compared with SpiderMonkey (the original JavaScript implementation in Firefox), SquirrelFish Extreme (the JavaScript implementation in Safari), and V8 (the JavaScript implementation in Chrome). Figure 11 summarizes the results. TraceMonkey was the fastest on 9 out of the 26

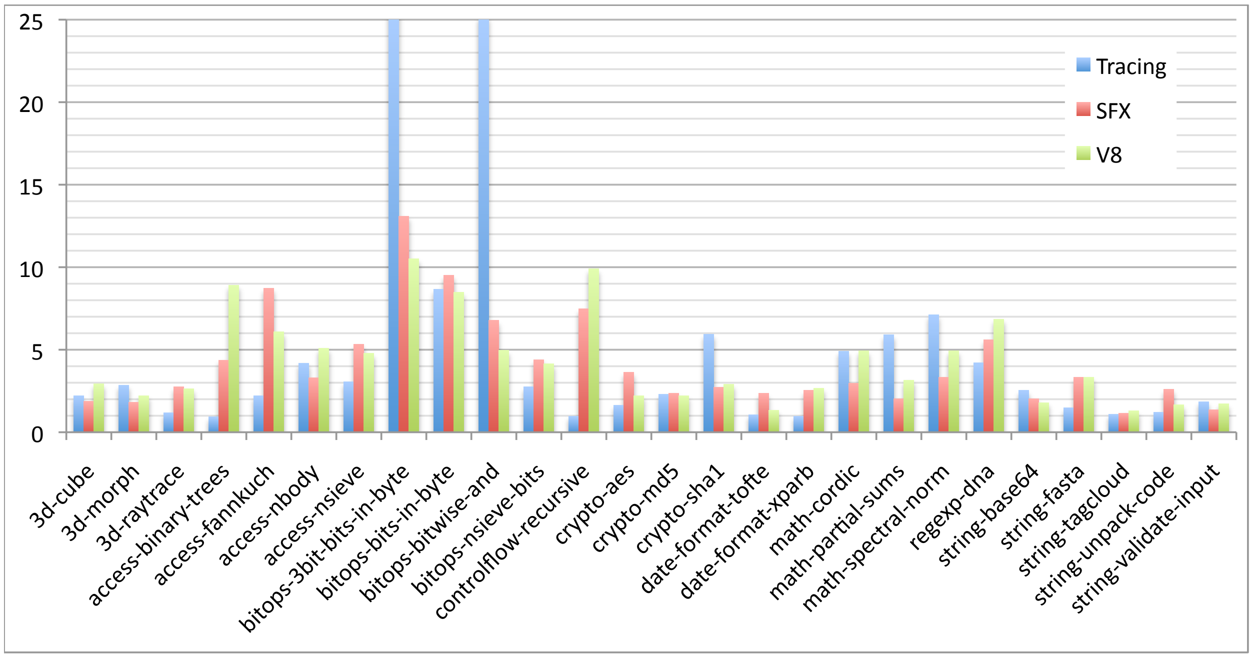

Figure 11: Performance of TraceMonkey, compared to SquirrelFish Extreme and V8. The baseline is SpiderMonkey. [12, Fig. 10]

benchmarks, and performed best on benchmarks that involved integer and bitwise operations. As mentioned before, this is because JavaScript integers are boxed and represented as floats, but TraceMonkey is able to generate specialized code.

However, TraceMonkey performed poorly with regular expressions, recursion, `eval`, and code that called C functions. Unlike most JavaScript engines, TraceMonkey does not have a special implementation for regular expressions. Furthermore, TraceMonkey has difficulty tracing recursion, and cannot trace `eval` or C functions. In addition to these shortcomings, TraceMonkey had an extremely high overhead for tracing. The authors conducted some experiments and estimated that recording and compiling traces incurred a 200x slowdown.

X: Note that these benchmarks have *nothing* at all to do with web programming. We have bitwise operations, crypto, date formatting, ray tracing, and silly numeric programs from the 1960s that have been ported from other systems. These are not representative of real web code.

MHY: There was a paper in 2011 by Richards, Gal, Eich, and Vitek [19], on finding more representative JavaScript benchmarks.

X: They found that tracing JITs did not work well on real web code. The hypothesis is that there isn't enough stability in the code.

Y: If you find a paper like this while doing your readings, please tell us! Dead ends in research are very common and interesting, but rarely mentioned. The important thing is to ask, is this a real dead end, or can we take a new approach?

Eventually, TraceMonkey was replaced by JägerMonkey, a method-based JIT. Tracing had incredibly high overhead, and transitioning between the interpreted code and compiled code was very expensive. It seems the original plan was for JägerMonkey to trace, with a fallback to method-based compilation, but it is not clear if this was ever implemented. Eventually, other techniques for type inference improved significantly, and TraceMonkey was completely removed from Firefox. Today, Firefox's JavaScript implementation is called IonMonkey.

X: Note that tracing is not inherently bad. It just didn't work very well for JavaScript.

MHY: It seems to be much more successful with PyPy and LuaJIT.

LuaJIT [16] seems to be quite successful, but there are no publications on it. The maintainer, Mike Pall, has open-sourced all his code, but he prefers to spend his time developing LuaJIT instead of writing papers.

X: The Lua community is... different. They don't cultivate papers. But it's a small language. Anyone can improve it, and they do.

Y: Kind of like Scheme before R4.

X: Are you an old man?

Y: What's that? Speak up.

4 PyPy

Tracing the Meta-Level: PyPy's Tracing JIT Compiler
Bolz, Cuni, Fijałkowski, and Rigo. ICOOOLPS 2009. [7]

Around the same time as TraceMonkey, the PyPy project also explored tracing to improve performance. PyPy [5] is a toolchain for implementing dynamic languages in RPython, a subset of Python that can easily be compiled. The dynamic language implementer can take advantage of the high-level features of RPython, without worrying about low-level details such as memory management, object layout, or threading models.

X: What is the relation between PyPy and Python?

MHY: The interpreter is written in RPython, a subset of Python. The original goal was to implement Python in RPython, hence PyPy. But now it's general framework for implementing dynamic languages.

Y: Pycket [4] is implemented in PyPy, basically implementing the CEK machine in RPython. They got extremely good performance for the subset of the language they implemented.

The authors of PyPy wanted to improve the performance of *all* interpreters implemented with their toolchain. Thus, the idea is to trace the *interpreter*, rather than the user program. In this sense, the main inspiration comes from DynamoRIO [22], which was a project that applied Dynamo's tracing technique to interpreters.

Because this work involves two interpreters, the terminology can be a little confusing. The PyPy authors use *language interpreter* to refer to the interpreter written by the language implementer (which executes user programs). On the other hand, the *tracing interpreter* is part of PyPy, and it traces the language interpreter.

Figure 12 shows a simple bytecode interpreter implemented in RPython. The majority of the `interpret` method is the bytecode dispatch loop. On line 5, it fetches and decodes the next instruction. On line 6, it increments the program counter, `pc`. Then it executes the instruction.

The language has an accumulator register `a`, 256 general-purpose registers, and six instructions. `JUMP_IF_A` decodes the target and jumps to it if `a` is non-zero. `MOV_A_R` decodes the destination register `n` and moves the contents of `a` to `n`. `MOV_R_A` moves the contents of register `n` to the accumulator. `ADD_R_TO_A` adds the value of register `n` to the accumulator. `DECR_A` decrements the accumulator by 1. Finally, `RETURN_A` returns the value of the accumulator.

Figure 13 is a bytecode program that computes the square of the accumulator. The precise details of the program are not necessary for the discussion, but note that lines 6 to 16 constitute the hot loop of the program.

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                pc = target
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = a
        elif opcode == MOV_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == ADD_R_TO_A:
            n = ord(bytecode[pc])
            pc += 1
            a += regs[n]
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

Figure 12: A simple bytecode interpreter written in RPython. The bytecode consists of six instructions, which access an accumulator register a and 256 general-purpose registers. [7, Fig. 2]

```
    # compute the square of the accumulator

    MOV_A_R     0   # i = a
    MOV_A_R     1   # copy of a

    # 4:
    MOV_R_A     0   # i--
    DECR_A
    MOV_A_R     0

    MOV_R_A     2   # res += a
    ADD_R_TO_A  1
    MOV_A_R     2

    MOV_R_A     0   # if i != 0: goto 4
    JUMP_IF_A   4

    MOV_R_A     2   # return res
    RETURN_A
```

Figure 13: Example bytecode program that computes the square of the accumulator. [7, Fig. 3]

Naïvely tracing the interpreter is not enough, since a trace of the dispatch loop corresponds to executing a single bytecode instruction. This is unhelpful, since executing a different instruction requires a new trace. We would like to "unroll" the dispatch loop so that it corresponds to a loop in the user program. Such a loop occurs if the program counter of the language interpreter takes on a previous value. In other words, a cycle is detected if a trace returns to the same position in the language interpreter, with the same interpreter program counter.

The language implementer must provide hints to PyPy, to indicate which variables represent the program counter and when backward jumps can occur. In Figure 12, the program counter is represented by the `pc` and `bytecode` variables, and backward jumps can occur at line 11.

X: This seems to work nicely for loops, but what about recursion?

Y: There are old techniques for turning programs into loops.

Z: See the Pycket paper [4].

However, this approach is still not enough. Consider the first instruction in the user program's loop, line 7 of Figure 13. Tracing the language interpreter's execution of only that instruction gives us

```
# MOV_R_A 0
opcode = ord(bytecode[pc])
pc += 1
guard opcode == MOV_R_A
n = ord(bytecode[pc])
pc += 1
a = regs[n]
```

Most of these instructions simply manipulate the state of the language interpreter, and only the last instruction represents the behaviour of the user program. However, note that `pc` and `bytecode` have some fixed values—they were checked before entering the trace. Therefore, the operations involving `pc` and `bytecode` can be constant folded, which gives us the following trace:

```
# MOV_R_A 0
a = regs[0]
```

The result is a trace of the language interpreter that resembles a trace of the user program.

The preliminary evaluation of PyPy involved simple benchmarks. One benchmark was the interpreter and bytecode program in Figures 12 and 13. The other benchmark was Python running a very small function. On that benchmark, PyPy performed faster than CPython, the default Python implementation. A later journal paper [6] provided better benchmarks.

X: Keep in mind that CPython is interpreted, so this is a low bar.

5 Loop-Aware Optimizations

Loop-Aware Optimizations in PyPy's Tracing JIT
Ardö, Bolz, and Fijałkowski. DLS 2012. [1]

Mike Pall originally implemented this optimization in LuaJIT. There are no publications on LuJIT, but Pall posted to the Lua mailing list [17] and invited researchers to explore and learn from his code.

> X: Why is Pall not one of the authors, then?
>
> MHY: He got a special acknowledgement, and reviewed drafts of the paper.
>
> Y: Who has heard of Schönfinkel? Once he gave a talk and someone—nobody knows who—in the audience took excellent notes, turned those into a paper, and published it under Schönfinkel's name. *That* is how you give someone credit.

Dynamic languages often have loop invariant code, which performs operations such as type checking, wrapping or unwrapping values, and method dispatch. The paper provides an even simpler example, which prints in an infinite loop:

```
L0(i0):
i1 = i0 + 1
print(i1)
jump(L0, i0)
```

In this code, `i0` is loop invariant. Therefore, the addition statement can be hoisted. However, optimizations in tracing JITs are designed to be very simple and fast, and therefore have no knowledge of control flow. Many optimizations are implemented as a single pass, and cannot determine `i0` is loop invariant.

However, if an earlier phase "peels" off a single iteration of the loop and duplicates it, the optimizer can see two separate iterations. Note that jumps and labels need to be updated:

```
# preamble
L0(i0):
i1 = i0 + 1
print(i1)
jump(L1, i0)

# peeled loop
L1(i0):
i2 = i0 + 1
print(i2)
jump(L1, i0)
```

Now the optimizer observes that `i2` can be replaced by `i1`, and the addition in the peeled loop can be removed. However, the jumps and labels need to be adjusted again:

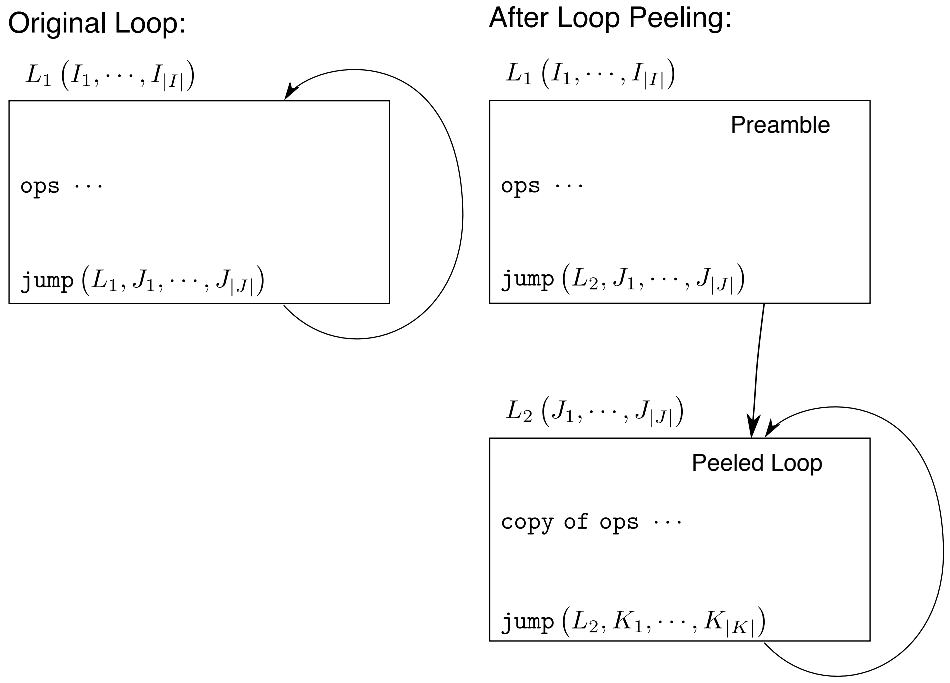
Original Loop:
$L_1\left(I_1, \cdots, I_{|I|}\right)$
ops ⋯
jump $\left(L_1, J_1, \cdots, J_{|J|}\right)$
After Loop Peeling:
$L_1\left(I_1, \cdots, I_{|I|}\right)$
Preamble
ops ⋯
jump $\left(L_2, J_1, \cdots, J_{|J|}\right)$
$L_2\left(J_1, \cdots, J_{|J|}\right)$
Peeled Loop
copy of ops ⋯
jump $\left(L_2, K_1, \cdots, K_{|K|}\right)$

Figure 14: Overview of loop peeling. [1, Fig. 3]

```
# preamble
L0(i0):
i1 = i0 + 1
print(i1)
jump(L1, i0)

# peeled loop
L1(i0, i1):
print(i1)
jump(L1, i0, i1)
```

Figure 14 depicts an overview of loop peeling. Note that the jump arguments of the original loop $(J_1, \ldots, J_{|J|})$ are equal to the label arguments of the peeled loop. Furthermore, the jump arguments of the preamble and the peeled loop need to be adjusted. The full details of this transformation, and how the mapping of variable renames is maintained, is described in the paper, along with a more realistic example and other optimizations that benefit from loop peeling.

The paper notes that TraceMonkey [12] implements a similar optimization for loop invariant code, but must directly implement it. In contrast, LuaJIT and PyPy are able to reuse existing optimization passes with just a simple loop peeling transformation.

The authors implemented loop peeling in PyPy in only 450 lines of code. They evaluated the technique with handwritten numeric benchmarks (such as convolution, dilation, edge finding, and square root), as well as the SciMark benchmarks [18]. They compared PyPy with and without loop peeling and LuaJIT with and without loop peeling. The authors observed a speedup of 70% with loop peeling. Figure 15 summarizes the results.

> X: One thing that these papers don't mention is that you need a very long warmup time before getting good numbers. It's a bit iffy because you usually don't want a along warmup time.
>
> Y: This is a point in favour for Pycket. It's OK to run the interpreter hundreds of times.
>
> X: What is the right methodology to study a self-adapting system? On the first run, you're just going to measure the compilation times. And maybe after the first hundred iterations, it's still not at the steady state.
>
> Z: Has anyone done a study on how long it takes to warmup?
>
> X: Yes, there was some work recently [3]. They had some really fun results. Sometimes there is no steady state and it just ping pongs!

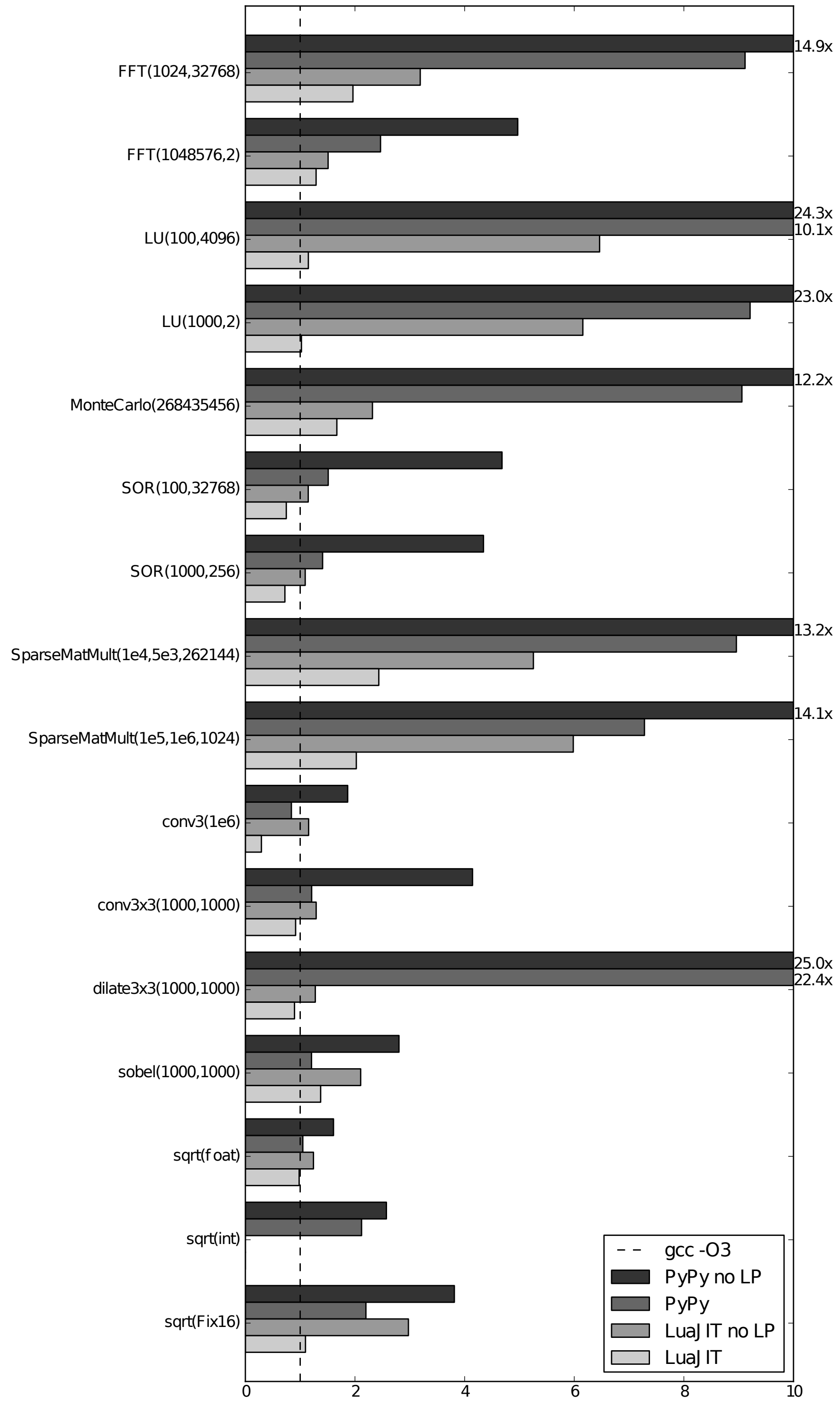

Figure 15: Benchmark results of loop peeling. The baseline is C, compiled with `gcc -O3`. [1, Fig. 10]

Conclusion

Tracing first appeared in 2000, in the Dynamo project. The idea was simple: identify hot traces of code in a program, record the sequence of instructions that are executed, and generate more efficient code. Since a trace is a linear sequence of code, it improves cache locality. Furthermore, the lack of internal control flow merge points means a trace is easier to optimize.

A few years later, tracing was implemented in a JIT for Java. Although the prototype did not match HotSpot's performance, it had significantly less overhead and required significantly less engineering effort.

There were hopes that tracing would be useful for JavaScript, a language that lacks static type information. TraceMonkey was a tracing JIT that speculates and types and generates specialized code. However, this ended up as a dead end: while TraceMonkey performed extremely well on integer benchmarks, it did not work well with real web code.

But tracing itself was not a dead end. With PyPy, tracing happens at the "meta-level," where the program being traced is an interpreter. LuaJIT is a successful tracing JIT for Lua, but there are no publications. However, the project is open source and available for researchers to examine. In particular, an optimization technique called "loop peeling" was first implemented in LuaJIT, but then adapted for PyPy and published.

Although TraceMonkey was not very successful and was eventually abandoned, tracing continues in LuaJIT and PyPy, and both projects are still being actively developed.

References

[1] H. Ardö, C. F. Bolz, and M. Fijałkowski. Loop-Aware Optimizations in PyPy's Tracing JIT. In *Proc. Dynamic Languages Symposium (DLS)*, 2012. DOI 10.1145/2384577.2384586.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. Programming Language Design and Implementation (PLDI)*, 2000. DOI 10.1145/349299.349303.

[3] E. Barett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt. Virtual Machine Warmup Blows Hot and Cold. Preprint. URL https://arxiv.org/abs/1602.00602, 2016.

[4] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A Tracing JIT for a Functional Language. In *Proc. International Conference on Functional Programming (ICFP)*, 2015. DOI 10.1145/2784731.2784740 URL https://github.com/pycket/pycket.

[5] C. F. Bolz and A. Rigo. How to not write a Virtual Machine. In *Proc. Dynamic Languages and Applications (DYLA)*, 2007. URL http://stups.hhu.de/mediawiki/images/7/7b/Pub-BoRi07_223.pdf.

[6] C. F. Bolz and L. Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 2015. DOI 10.1016/j.scico.2013.02.001.

[7] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proc. Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, 2009. DOI 10.1145/1565824.1565827.

[8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Principles of Programming Languages (POPL)*, 1977. DOI 10.1145/512950.512973.

[9] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. DOI 10.1145/2451116.2451141.

[10] A. Gal and M. Franz. Incremental Dynamic Code Generation with Trace Trees. Technical Report ICS-TR-006-16, University of California, Irvine, 2006. URL https://github.com/nuprl/hopl-s2017/raw/master/tracing-jit/Gal06_Trace_Trees.pdf.

[11] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proc. Virtual Execution Environments (VEE)*, 2006. DOI https://doi.org/10.1145/1134760.1134780.

[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. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic

Languages. In *Proc. Programming Language Design and Implementation (PLDI)*, 2009. DOI `10.1145/1542476.1542528`.

[13] D. Knuth. Structured Programming with `go to` Statements. *ACM Computing Surveys*, 1974. DOI `10.1145/356635.356640`.

[14] R. Lougher. JamVM — A compact Java Virtual Machine, 2003. URL `http://jamvm.sourceforge.net/`.

[15] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and Development of the Java Grande Benchmarks. In *Proc. ACM Conference on Java Grande (JAVA)*, 1999. DOI `10.1145/304065.304101`.

[16] M. Pall. The LuaJIT Project, 2005. URL `http://luajit.org/`.

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

[18] R. Pozo and B. Miller. SciMark 2.0, 2004. URL `http://math.nist.gov/scimark2/`.

[19] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated Construction of JavaScript Benchmarks. In *Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2011. DOI `10.1145/2048066.2048119`.

[20] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way Versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software Practice and Experience*, 1993. DOI `10.1002/spe.4380230507`.

[21] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks, 1995. URL `https://www.spec.org/cpu95/`.

[22] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic Native Optimization of Interpreters. In *Proc. Interpreters, Virtual Machines and Emulators (IVME)*, 2003. DOI `10.1145/858570.858576`.

[23] M. Wand. The Theory of Fexprs is Trivial. *Lisp Symbolic Computing*, 1998. DOI `10.1023/A:1007720632734`.

[24] WebKit Open Source Project. SunSpider JavaScript Benchmark, 2007. URL `https://webkit.org/perf/sunspider/sunspider.html`.