Hence, recording and compiling a trace *speculates* that the path and typing will be exactly as they were during recording for subsequent iterations of the loop.

Every compiled trace contains all the *guards* (checks) required to validate the speculation. If one of the guards fails (if control flow is different, or a value of a different type is generated), the trace exits. If an exit becomes hot, the VM can record a *branch trace* starting at the exit to cover the new path. In this way, the VM records a *trace tree* covering all the hot paths through the loop.

Nested loops can be difficult to optimize for tracing VMs. In a naïve implementation, inner loops would become hot first, and the VM would start tracing there. When the inner loop exits, the VM would detect that a different branch was taken. The VM would try to record a branch trace, and find that the trace reaches not the inner loop header, but the outer loop header. At this point, the VM could continue tracing until it reaches the inner loop header again, thus tracing the outer loop inside a trace tree for the inner loop. But this requires tracing a copy of the outer loop for every side exit and type combination in the inner loop. In essence, this is a form of unintended tail duplication, which can easily overflow the code cache. Alternatively, the VM could simply stop tracing, and give up on ever tracing outer loops.

We solve the nested loop problem by recording *nested trace trees*. Our system traces the inner loop exactly as the naïve version. The system stops extending the inner tree when it reaches an outer loop, but then it starts a new trace at the outer loop header. When the outer loop reaches the inner loop header, the system tries to call the trace tree for the inner loop. If the call succeeds, the VM records the call to the inner tree as part of the outer trace and finishes the outer trace as normal. In this way, our system can trace any number of loops nested to any depth without causing excessive tail duplication.

These techniques allow a VM to dynamically translate a program to nested, type-specialized trace trees. Because traces can cross function call boundaries, our techniques also achieve the effects of inlining. Because traces have no internal control-flow joins, they can be optimized in linear time by a simple compiler (10). Thus, our tracing VM efficiently performs the same kind of optimizations that would require interprocedural analysis in a static optimization setting. This makes tracing an attractive and effective tool to type specialize even complex function call-rich code.

We implemented these techniques for an existing JavaScript interpreter, SpiderMonkey. We call the resulting tracing VM *TraceMonkey*. TraceMonkey supports all the JavaScript features of SpiderMonkey, with a 2x-20x speedup for traceable programs.

This paper makes the following contributions:

- We explain an algorithm for dynamically forming trace trees to cover a program, representing nested loops as nested trace trees.
- We explain how to speculatively generate efficient type-specialized code for traces from dynamic language programs.
- We validate our tracing techniques in an implementation based on the SpiderMonkey JavaScript interpreter, achieving 2x-20x speedups on many programs.

The remainder of this paper is organized as follows. Section 3 is a general overview of trace tree based compilation we use to capture and compile frequently executed code regions. In Section 4 we describe our approach of covering nested loops using a number of individual trace trees. In Section 5 we describe our trace-compilation based speculative type specialization approach we use to generate efficient machine code from recorded bytecode traces. Our implementation of a dynamic type-specializing compiler for JavaScript is described in Section 6. Related work is discussed in Section 8. In Section 7 we evaluate our dynamic compiler based on a set of industry benchmarks. The paper ends with conclusions in Section 9 and an outlook on future work is presented in Section 10.

```
1 for (var i = 2; i < 100; ++i) {
2   if (!primes[i])
3     continue;
4   for (var k = i + i; i < 100; k += i)
5     primes[k] = false;
6 }
```

Figure 1. Sample program: sieve of Eratosthenes. `primes` is initialized to an array of 100 `false` values on entry to this code snippet.

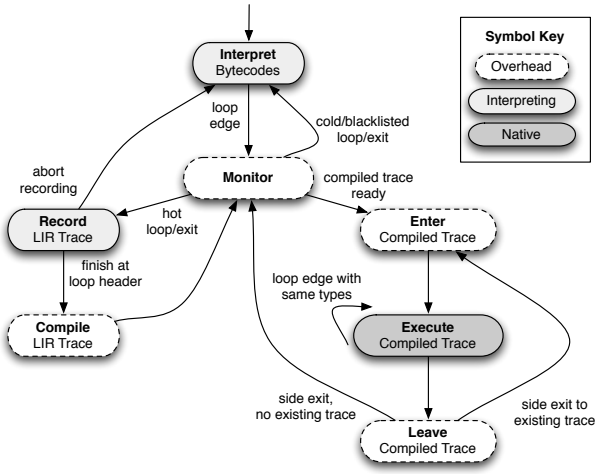

Figure 2. State machine describing the major activities of TraceMonkey and the conditions that cause transitions to a new activity. In the dark box, TM executes JS as compiled traces. In the light gray boxes, TM executes JS in the standard interpreter. White boxes are overhead. Thus, to maximize performance, we need to maximize time spent in the darkest box and minimize time spent in the white boxes. The best case is a loop where the types at the loop edge are the same as the types on entry–then TM can stay in native code until the loop is done.

2. Overview: Example Tracing Run

This section provides an overview of our system by describing how TraceMonkey executes an example program. The example program, shown in Figure 1, computes the first 100 prime numbers with nested loops. The narrative should be read along with Figure 2, which describes the activities TraceMonkey performs and when it transitions between the loops.

TraceMonkey always begins executing a program in the bytecode interpreter. Every loop back edge is a potential trace point. When the interpreter crosses a loop edge, TraceMonkey invokes the *trace monitor*, which may decide to record or execute a native trace. At the start of execution, there are no compiled traces yet, so the trace monitor counts the number of times each loop back edge is executed until a loop becomes *hot*, currently after 2 crossings. Note that the way our loops are compiled, the loop edge is crossed before entering the loop, so the second crossing occurs immediately after the first iteration.

Here is the sequence of events broken down by outer loop iteration: