i=4. On this iteration, TraceMonkey calls T_{16}. Because `i=4`, the `if` statement on line 2 is taken. This branch was not taken in the original trace, so this causes T_{16} to fail a guard and take a side exit. The exit is not yet hot, so TraceMonkey returns to the interpreter, which executes the continue statement.

i=5. TraceMonkey calls T_{16}, which in turn calls the nested trace T_{45}. T_{16} loops back to its own header, starting the next iteration without ever returning to the monitor.

i=6. On this iteration, the side exit on line 2 is taken again. This time, the side exit becomes hot, so a trace $T_{23,1}$ is recorded that covers line 3 and returns to the loop header. Thus, the end of $T_{23,1}$ jumps directly to the start of T_{16}. The side exit is patched so that on future iterations, it jumps directly to $T_{23,1}$.

At this point, TraceMonkey has compiled enough traces to cover the entire nested loop structure, so the rest of the program runs entirely as native code.

3. Trace Trees

In this section, we describe traces, trace trees, and how they are formed at run time. Although our techniques apply to any dynamic language interpreter, we will describe them assuming a bytecode interpreter to keep the exposition simple.

3.1 Traces

A *trace* is simply a program path, which may cross function call boundaries. TraceMonkey focuses on *loop traces*, that originate at a loop edge and represent a single iteration through the associated loop.

Similar to an extended basic block, a trace is only entered at the top, but may have many exits. In contrast to an extended basic block, a trace can contain join nodes. Since a trace always only follows one single path through the original program, however, join nodes are not recognizable as such in a trace and have a single predecessor node like regular nodes.

A *typed trace* is a trace annotated with a type for every variable (including temporaries) on the trace. A typed trace also has an entry *type map* giving the required types for variables used on the trace before they are defined. For example, a trace could have a type map `(x: int, b: boolean)`, meaning that the trace may be entered only if the value of the variable `x` is of type `int` and the value of `b` is of type `boolean`. The entry type map is much like the signature of a function.

In this paper, we only discuss typed loop traces, and we will refer to them simply as "traces". The key property of typed loop traces is that they can be compiled to efficient machine code using the same techniques used for typed languages.

In TraceMonkey, traces are recorded in trace-flavored SSA *LIR* (low-level intermediate representation). In trace-flavored SSA (or TSSA), phi nodes appear only at the entry point, which is reached both on entry and via loop edges. The important LIR primitives are constant values, memory loads and stores (by address and offset), integer operators, floating-point operators, function calls, and conditional exits. Type conversions, such as integer to double, are represented by function calls. This makes the LIR used by TraceMonkey independent of the concrete type system and type conversion rules of the source language. The LIR operations are generic enough that the backend compiler is language independent. Figure 3 shows an example LIR trace.

Bytecode interpreters typically represent values in a various complex data structures (e.g., hash tables) in a boxed format (i.e., with attached type tag bits). Since a trace is intended to represent efficient code that eliminates all that complexity, our traces operate on unboxed values in simple variables and arrays as much as possible.

A trace records all its intermediate values in a small activation record area. To make variable accesses fast on trace, the trace also imports local and global variables by unboxing them and copying them to its activation record. Thus, the trace can read and write these variables with simple loads and stores from a native activation recording, independently of the boxing mechanism used by the interpreter. When the trace exits, the VM boxes the values from this native storage location and copies them back to the interpreter structures.

For every control-flow branch in the source program, the recorder generates conditional exit LIR instructions. These instructions exit from the trace if required control flow is different from what it was at trace recording, ensuring that the trace instructions are run only if they are supposed to. We call these instructions *guard* instructions.

Most of our traces represent loops and end with the special `loop` LIR instruction. This is just an unconditional branch to the top of the trace. Such traces return only via guards.

Now, we describe the key optimizations that are performed as part of recording LIR. All of these optimizations reduce complex dynamic language constructs to simple typed constructs by specializing for the current trace. Each optimization requires guard instructions to verify their assumptions about the state and exit the trace if necessary.

Type specialization.

All LIR primitives apply to operands of specific types. Thus, LIR traces are necessarily type-specialized, and a compiler can easily produce a translation that requires no type dispatches. A typical bytecode interpreter carries tag bits along with each value, and to perform any operation, must check the tag bits, dynamically dispatch, mask out the tag bits to recover the untagged value, perform the operation, and then reapply tags. LIR omits everything except the operation itself.

A potential problem is that some operations can produce values of unpredictable types. For example, reading a property from an object could yield a value of any type, not necessarily the type observed during recording. The recorder emits guard instructions that conditionally exit if the operation yields a value of a different type from that seen during recording. These guard instructions guarantee that as long as execution is on trace, the types of values match those of the typed trace. When the VM observes a side exit along such a type guard, a new typed trace is recorded originating at the side exit location, capturing the new type of the operation in question.

Representation specialization: objects. In JavaScript, name lookup semantics are complex and potentially expensive because they include features like object inheritance and `eval`. To evaluate an object property read expression like `o.x`, the interpreter must search the property map of `o` and all of its prototypes and parents. Property maps can be implemented with different data structures (e.g., per-object hash tables or shared hash tables), so the search process also must dispatch on the representation of each object found during search. TraceMonkey can simply observe the result of the search process and record the simplest possible LIR to access the property value. For example, the search might finds the value of `o.x` in the prototype of `o`, which uses a shared hash-table representation that places `x` in slot 2 of a property vector. Then the recorded can generate LIR that reads `o.x` with just two or three loads: one to get the prototype, possibly one to get the property value vector, and one more to get slot 2 from the vector. This is a vast simplification and speedup compared to the original interpreter code. Inheritance relationships and object representations can change during execution, so the simplified code requires guard instructions that ensure the object representation is the same. In TraceMonkey, objects' rep-