resentations are assigned an integer key called the *object shape*. Thus, the guard is a simple equality check on the object shape.

Representation specialization: numbers. JavaScript has no integer type, only a Number type that is the set of 64-bit IEEE-754 floating-pointer numbers ("doubles"). But many JavaScript operators, in particular array accesses and bitwise operators, really operate on integers, so they first convert the number to an integer, and then convert any integer result back to a double.[1] Clearly, a JavaScript VM that wants to be fast must find a way to operate on integers directly and avoid these conversions.

In TraceMonkey, we support two representations for numbers: integers and doubles. The interpreter uses integer representations as much as it can, switching for results that can only be represented as doubles. When a trace is started, some values may be imported and represented as integers. Some operations on integers require guards. For example, adding two integers can produce a value too large for the integer representation.

Function inlining. LIR traces can cross function boundaries in either direction, achieving function inlining. Move instructions need to be recorded for function entry and exit to copy arguments in and return values out. These move statements are then optimized away by the compiler using copy propagation. In order to be able to return to the interpreter, the trace must also generate LIR to record that a call frame has been entered and exited. The frame entry and exit LIR saves just enough information to allow the intepreter call stack to be restored later and is much simpler than the interpreter's standard call code. If the function being entered is not constant (which in JavaScript includes any call by function name), the recorder must also emit LIR to guard that the function is the same.

Guards and side exits. Each optimization described above requires one or more guards to verify the assumptions made in doing the optimization. A guard is just a group of LIR instructions that performs a test and conditional exit. The exit branches to a *side exit*, a small off-trace piece of LIR that returns a pointer to a structure that describes the reason for the exit along with the interpreter PC at the exit point and any other data needed to restore the interpreter's state structures.

Aborts. Some constructs are difficult to record in LIR traces. For example, `eval` or calls to external functions can change the program state in unpredictable ways, making it difficult for the tracer to know the current type map in order to continue tracing. A tracing implementation can also have any number of other limitations, e.g.,a small-memory device may limit the length of traces. When any situation occurs that prevents the implementation from continuing trace recording, the implementation *aborts* trace recording and returns to the trace monitor.

3.2 Trace Trees

Especially simple loops, namely those where control flow, value types, value representations, and inlined functions are all invariant, can be represented by a single trace. But most loops have at least some variation, and so the program will take side exits from the main trace. When a side exit becomes hot, TraceMonkey starts a new *branch trace* from that point and patches the side exit to jump directly to that trace. In this way, a single trace expands on demand to a single-entry, multiple-exit *trace tree*.

This section explains how trace trees are formed during execution. The goal is to form trace trees during execution that cover all the hot paths of the program.

[1] Arrays are actually worse than this: if the index value is a number, it must be converted from a double to a string for the property access operator, and then to an integer internally to the array implementation.

Starting a tree. Tree trees always start at loop headers, because they are a natural place to look for hot paths. In TraceMonkey, loop headers are easy to detect–the bytecode compiler ensures that a bytecode is a loop header iff it is the target of a backward branch. TraceMonkey starts a tree when a given loop header has been executed a certain number of times (2 in the current implementation). Starting a tree just means starting recording a trace for the current point and type map and marking the trace as the root of a tree. Each tree is associated with a loop header and type map, so there may be several trees for a given loop header.

Closing the loop. Trace recording can end in several ways.

Ideally, the trace reaches the loop header where it started with the same type map as on entry. This is called a *type-stable* loop iteration. In this case, the end of the trace can jump right to the beginning, as all the value representations are exactly as needed to enter the trace. The jump can even skip the usual code that would copy out the state at the end of the trace and copy it back in to the trace activation record to enter a trace.

In certain cases the trace might reach the loop header with a different type map. This scenario is sometime observed for the first iteration of a loop. Some variables inside the loop might initially be *undefined*, before they are set to a concrete type during the first loop iteration. When recording such an iteration, the recorder cannot link the trace back to its own loop header since it is *type-unstable*. Instead, the iteration is terminated with a side exit that will always fail and return to the interpreter. At the same time a new trace is recorded with the new type map. Every time an additional type-unstable trace is added to a region, its exit type map is compared to the entry map of all existing traces in case they complement each other. With this approach we are able to cover type-unstable loop iterations as long they eventually form a stable equilibrium.

Finally, the trace might exit the loop before reaching the loop header, for example because execution reaches a `break` or `return` statement. In this case, the VM simply ends the trace with an exit to the trace monitor.

As mentioned previously, we may speculatively chose to represent certain Number-typed values as integers on trace. We do so when we observe that Number-typed variables contain an integer value at trace entry. If during trace recording the variable is unexpectedly assigned a non-integer value, we have to widen the type of the variable to a double. As a result, the recorded trace becomes inherently type-unstable since it starts with an integer value but ends with a double value. This represents a mis-speculation, since at trace entry we specialized the Number-typed value to an integer, assuming that at the loop edge we would again find an integer value in the variable, allowing us to close the loop. To avoid future speculative failures involving this variable, and to obtain a type-stable trace we note the fact that the variable in question as been observed to sometimes hold non-integer values in an advisory data structure which we call the "oracle".

When compiling loops, we consult the oracle before specializing values to integers. Speculation towards integers is performed only if no adverse information is known to the oracle about that particular variable. Whenever we accidentally compile a loop that is type-unstable due to mis-speculation of a Number-typed variable, we immediately trigger the recording of a new trace, which based on the now updated oracle information will start with a double value and thus become type stable.

Extending a tree. Side exits lead to different paths through the loop, or paths with different types or representations. Thus, to completely cover the loop, the VM must record traces starting at all side exits. These traces are recorded much like root traces: there is a counter for each side exit, and when the counter reaches a hotness threshold, recording starts. Recording stops exactly as for the root trace, using the loop header of the root trace as the target to reach.