Our implementation does not extend at all side exits. It extends only if the side exit is for a control-flow branch, and only if the side exit does not leave the loop. In particular we do not want to extend a trace tree along a path that leads to an outer loop, because we want to cover such paths in an outer tree through tree *nesting*.

3.3 Blacklisting

Sometimes, a program follows a path that cannot be compiled into a trace, usually because of limitations in the implementation. TraceMonkey does not currently support recording throwing and catching of arbitrary exceptions. This design trade off was chosen, because exceptions are usually rare in JavaScript. However, if a program opts to use exceptions intensively, we would suddenly incur a punishing runtime overhead if we repeatedly try to record a trace for this path and repeatedly fail to do so, since we abort tracing every time we observe an exception being thrown.

As a result, if a hot loop contains traces that always fail, the VM could potentially run much more slowly than the base interpreter: the VM repeatedly spends time trying to record traces, but is never able to run any. To avoid this problem, whenever the VM is about to start tracing, it must try to predict whether it will finish the trace.

Our prediction algorithm is based on *blacklisting* traces that have been tried and failed. When the VM fails to finish a trace starting at a given point, the VM records that a failure has occurred. The VM also sets a counter so that it will not try to record a trace starting at that point until it is passed a few more times (32 in our implementation). This *backoff* counter gives temporary conditions that prevent tracing a chance to end. For example, a loop may behave differently during startup than during its steady-state execution. After a given number of failures (2 in our implementation), the VM marks the fragment as blacklisted, which means the VM will never again start recording at that point.

After implementing this basic strategy, we observed that for small loops that get blacklisted, the system can spend a noticeable amount of time just finding the loop fragment and determining that it has been blacklisted. We now avoid that problem by patching the bytecode. We define an extra no-op bytecode that indicates a loop header. The VM calls into the trace monitor every time the interpreter executes a loop header no-op. To blacklist a fragment, we simply replace the loop header no-op with a regular no-op. Thus, the interpreter will never again even call into the trace monitor.

There is a related problem we have not yet solved, which occurs when a loop meets all of these conditions:

- The VM can form at least one root trace for the loop.
- There is at least one hot side exit for which the VM cannot complete a trace.
- The loop body is short.

In this case, the VM will repeatedly pass the loop header, search for a trace, find it, execute it, and fall back to the interpreter. With a short loop body, the overhead of finding and calling the trace is high, and causes performance to be even slower than the basic interpreter. So far, in this situation we have improved the implementation so that the VM can complete the branch trace. But it is hard to guarantee that this situation will never happen. As future work, this situation could be avoided by detecting and blacklisting loops for which the average trace call executes few bytecodes before returning to the interpreter.

4. Nested Trace Tree Formation

Figure 7 shows basic trace tree compilation (11) applied to a nested loop where the inner loop contains two paths. Usually, the inner loop (with header at i_2) becomes hot first, and a trace tree is rooted at that point. For example, the first recorded trace may be a cycle

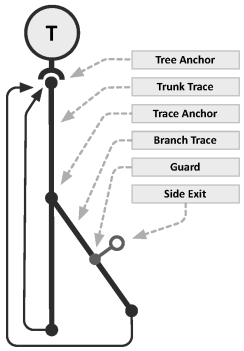


Figure 5. A tree with two traces, a trunk trace and one branch trace. The trunk trace contains a guard to which a branch trace was attached. The branch trace contain a guard that may fail and trigger a side exit. Both the trunk and the branch trace loop back to the tree anchor, which is the beginning of the trace tree.

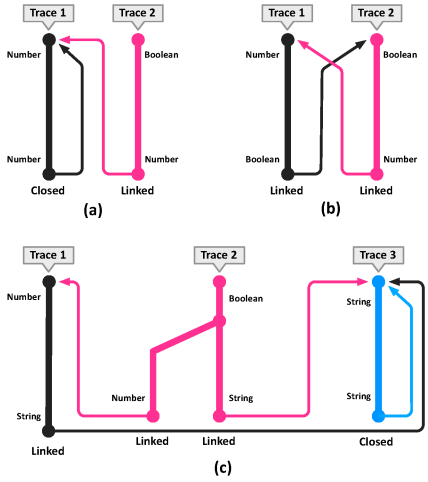


Figure 6. We handle type-unstable loops by allowing traces to compile that cannot loop back to themselves due to a type mismatch. As such traces accumulate, we attempt to connect their loop edges to form groups of trace trees that can execute without having to side-exit to the interpreter to cover odd type cases. This is particularly important for nested trace trees where an outer tree tries to call an inner tree (or in this case a forest of inner trees), since inner loops frequently have initially undefined values which change type to a concrete value after the first iteration.

through the inner loop, $\{i_2, i_3, i_5, \alpha\}$. The α symbol is used to indicate that the trace loops back the tree anchor.

When execution leaves the inner loop, the basic design has two choices. First, the system can stop tracing and give up on compiling the outer loop, clearly an undesirable solution. The other choice is to continue tracing, compiling traces for the outer loop inside the inner loop's trace tree.

For example, the program might exit at i_5 and record a branch trace that incorporates the outer loop: $\{i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Later, the program might take the other branch at i_2 and then exit, recording another branch trace incorporating the outer loop: $\{i_2, i_4, i_5, i_7, i_1, i_6, i_7, i_1, \alpha\}$. Thus, the outer loop is recorded and compiled twice, and both copies must be retained in the trace cache.