When a trace call returns, the monitor restores the interpreter state. First, the monitor checks the reason for the trace exit and applies blacklisting if needed. Then, it pops or synthesizes interpreter JavaScript call stack frames as needed. Finally, it copies the imported variables back from the trace activation record to the interpreter state.

At least in the current implementation, these steps have a non-negligible runtime cost, so minimizing the number of interpreter-to-trace and trace-to-interpreter transitions is essential for performance. (see also Section 3.3). Our experiments (see Figure 12) show that for programs we can trace well such transitions happen infrequently and hence do not contribute significantly to total runtime. In a few programs, where the system is prevented from recording branch traces for hot side exits by aborts, this cost can rise to up to 10% of total execution time.

6.2 Trace Stitching

Transitions from a trace to a branch trace at a side exit avoid the costs of calling traces from the monitor, in a feature called *trace stitching*. At a side exit, the exiting trace only needs to write live register-carried values back to its trace activation record. In our implementation, identical type maps yield identical activation record layouts, so the trace activation record can be reused immediately by the branch trace.

In programs with branchy trace trees with small traces, trace stitching has a noticeable cost. Although writing to memory and then soon reading back would be expected to have a high L1 cache hit rate, for small traces the increased instruction count has a noticeable cost. Also, if the writes and reads are very close in the dynamic instruction stream, we have found that current x86 processors often incur penalties of 6 cycles or more (e.g., if the instructions use different base registers with equal values, the processor may not be able to detect that the addresses are the same right away).

The alternate solution is to recompile an entire trace tree, thus achieving inter-trace register allocation (10). The disadvantage is that tree recompilation takes time quadratic in the number of traces. We believe that the cost of recompiling a trace tree every time a branch is added would be prohibitive. That problem might be mitigated by recompiling only at certain points, or only for very hot, stable trees.

In the future, multicore hardware is expected to be common, making background tree recompilation attractive. In a closely related project (13) background recompilation yielded speedups of up to 1.25x on benchmarks with many branch traces. We plan to apply this technique to TraceMonkey as future work.

6.3 Trace Recording

The job of the trace recorder is to emit LIR with identical semantics to the currently running interpreter bytecode trace. A good implementation should have low impact on non-tracing interpreter performance and a convenient way for implementers to maintain semantic equivalence.

In our implementation, the only direct modification to the interpreter is a call to the trace monitor at loop edges. In our benchmark results (see Figure 12) the total time spent in the monitor (for all activities) is usually less than 5%, so we consider the interpreter impact requirement met. Incrementing the loop hit counter is expensive because it requires us to look up the loop in the trace cache, but we have tuned our loops to become hot and trace very quickly (on the second iteration). The hit counter implementation could be improved, which might give us a small increase in overall performance, as well as more flexibility with tuning hotness thresholds. Once a loop is blacklisted we never call into the trace monitor for that loop (see Section 3.3).

Recording is activated by a pointer swap that sets the interpreter's dispatch table to call a single "interrupt" routine for every bytecode. The interrupt routine first calls a bytecode-specific recording routine. Then, it turns off recording if necessary (e.g., the trace ended). Finally, it jumps to the standard interpreter bytecode implementation. Some bytecodes have effects on the type map that cannot be predicted before executing the bytecode (e.g., calling `String.charCodeAt`, which returns an integer or *NaN* if the index argument is out of range). For these, we arrange for the interpreter to call into the recorder again after executing the bytecode. Since such hooks are relatively rare, we embed them directly into the interpreter, with an additional runtime check to see whether a recorder is currently active.

While separating the interpreter from the recorder reduces individual code complexity, it also requires careful implementation and extensive testing to achieve semantic equivalence.

In some cases achieving this equivalence is difficult since SpiderMonkey follows a *fat-bytecode* design, which was found to be beneficial to pure interpreter performance.

In fat-bytecode designs, individual bytecodes can implement complex processing (e.g., the `getprop` bytecode, which implements full JavaScript property value access, including special cases for cached and dense array access).

Fat bytecodes have two advantages: fewer bytecodes means lower dispatch cost, and bigger bytecode implementations give the compiler more opportunities to optimize the interpreter.

Fat bytecodes are a problem for TraceMonkey because they require the recorder to reimplement the same special case logic in the same way. Also, the advantages are reduced because (a) dispatch costs are eliminated entirely in compiled traces, (b) the traces contain only one special case, not the interpreter's large chunk of code, and (c) TraceMonkey spends less time running the base interpreter.

One way we have mitigated these problems is by implementing certain complex bytecodes in the recorder as sequences of simple bytecodes. Expressing the original semantics this way is not too difficult, and recording simple bytecodes is much easier. This enables us to retain the advantages of fat bytecodes while avoiding some of their problems for trace recording. This is particularly effective for fat bytecodes that recurse back into the interpreter, for example to convert an object into a primitive value by invoking a well-known method on the object, since it lets us inline this function call.

It is important to note that we split fat opcodes into thinner opcodes only during recording. When running purely interpretatively (i.e. code that has been blacklisted), the interpreter directly and efficiently executes the fat opcodes.

6.4 Preemption

SpiderMonkey, like many VMs, needs to preempt the user program periodically. The main reasons are to prevent infinitely looping scripts from locking up the host system and to schedule GC.

In the interpreter, this had been implemented by setting a "preempt now" flag that was checked on every backward jump. This strategy carried over into TraceMonkey: the VM inserts a guard on the preemption flag at every loop edge. We measured less than a 1% increase in runtime on most benchmarks for this extra guard. In practice, the cost is detectable only for programs with very short loops.

We tested and rejected a solution that avoided the guards by compiling the loop edge as an unconditional jump, and patching the jump target to an exit routine when preemption is required. This solution can make the normal case slightly faster, but then preemption becomes very slow. The implementation was also very complex, especially trying to restart execution after the preemption.