5.1 Optimizations

Because traces are in SSA form and have no join points or ϕ-nodes, certain optimizations are easy to implement. In order to get good startup performance, the optimizations must run quickly, so we chose a small set of optimizations. We implemented the optimizations as pipelined filters so that they can be turned on and off independently, and yet all run in just two loop passes over the trace: one forward and one backward.

Every time the trace recorder emits a LIR instruction, the instruction is immediately passed to the first filter in the forward pipeline. Thus, forward filter optimizations are performed as the trace is recorded. Each filter may pass each instruction to the next filter unchanged, write a different instruction to the next filter, or write no instruction at all. For example, the constant folding filter can replace a multiply instruction like $v_{13} := mul3, 1000$ with a constant instruction $v_{13} = 3000$.

We currently apply four forward filters:

- On ISAs without floating-point instructions, a soft-float filter converts floating-point LIR instructions to sequences of integer instructions.
- CSE (constant subexpression elimination),
- expression simplification, including constant folding and a few algebraic identities (e.g., $a - a = 0$), and
- source language semantic-specific expression simplification, primarily algebraic identities that allow DOUBLE to be replaced with INT. For example, LIR that converts an INT to a DOUBLE and then back again would be removed by this filter.

When trace recording is completed, nanojit runs the backward optimization filters. These are used for optimizations that require backward program analysis. When running the backward filters, nanojit reads one LIR instruction at a time, and the reads are passed through the pipeline.

We currently apply three backward filters:

- Dead data-stack store elimination. The LIR trace encodes many stores to locations in the interpreter stack. But these values are never read back before exiting the trace (by the interpreter or another trace). Thus, stores to the stack that are overwritten before the next exit are dead. Stores to locations that are off the top of the interpreter stack at future exits are also dead.
- Dead call-stack store elimination. This is the same optimization as above, except applied to the interpreter's call stack used for function call inlining.
- Dead code elimination. This eliminates any operation that stores to a value that is never used.

After a LIR instruction is successfully read ("pulled") from the backward filter pipeline, nanojit's code generator emits native machine instruction(s) for it.

5.2 Register Allocation

We use a simple greedy register allocator that makes a single backward pass over the trace (it is integrated with the code generator). By the time the allocator has reached an instruction like $v_3 = add\ v_1, v_2$, it has already assigned a register to v_3. If v_1 and v_2 have not yet been assigned registers, the allocator assigns a free register to each. If there are no free registers, a value is selected for spilling. We use a class heuristic that selects the "oldest" register-carried value (6).

The heuristic considers the set R of values v in registers immediately after the current instruction for spilling. Let v_m be the last instruction before the current where each v is referred to. Then the heuristic selects v with minimum v_m. The motivation is that this frees up a register for as long as possible given a single spill.

If we need to spill a value v_s at this point, we generate the restore code just after the code for the current instruction. The corresponding spill code is generated just after the last point where v_s was used. The register that was assigned to v_s is marked free for the preceding code, because that register can now be used freely without affecting the following code

Tag	JS Type	Description
xx1	number	31-bit integer representation
000	object	pointer to JSObject handle
010	number	pointer to double handle
100	string	pointer to JSString handle
110	boolean	enumeration for null, undefined, true, false
	null, or undefined	

Figure 9. Tagged values in the SpiderMonkey JS interpreter. Testing tags, unboxing (extracting the untagged value) and boxing (creating tagged values) are significant costs. Avoiding these costs is a key benefit of tracing.

6. Implementation

To demonstrate the effectiveness of our approach, we have implemented a trace-based dynamic compiler for the SpiderMonkey JavaScript Virtual Machine (4). SpiderMonkey is the JavaScript VM embedded in Mozilla's Firefox open-source web browser (2), which is used by more than 200 million users world-wide. The core of SpiderMonkey is a bytecode interpreter implemented in C++.

In SpiderMonkey, all JavaScript values are represented by the type `jsval`. A `jsval` is machine word in which up to the 3 of the least significant bits are a type tag, and the remaining bits are data. See Figure 6 for details. All pointers contained in `jsvals` point to GC-controlled blocks aligned on 8-byte boundaries.

JavaScript *object* values are mappings of string-valued property names to arbitrary values. They are represented in one of two ways in SpiderMonkey. Most objects are represented by a shared structural description, called the *object shape*, that maps property names to array indexes using a hash table. The object stores a pointer to the shape and the array of its own property values. Objects with large, unique sets of property names store their properties directly in a hash table.

The garbage collector is an exact, non-generational, stop-the-world mark-and-sweep collector.

In the rest of this section we discuss key areas of the TraceMonkey implementation.

6.1 Calling Compiled Traces

Compiled traces are stored in a *trace cache*, indexed by intepreter PC and type map. Traces are compiled so that they may be called as functions using standard native calling conventions (e.g., `FASTCALL` on x86).

The interpreter must hit a loop edge and enter the monitor in order to call a native trace for the first time. The monitor computes the current type map, checks the trace cache for a trace for the current PC and type map, and if it finds one, executes the trace.

To execute a trace, the monitor must build a trace activation record containing imported local and global variables, temporary stack space, and space for arguments to native calls. The local and global values are then copied from the interpreter state to the trace activation record. Then, the trace is called like a normal C function pointer.