

Home  
Reading  
Searching  
Subscribe  
Sponsors  
Statistics  
Posting  
Contact  
Spam  
Lists  
Links  
About  
Hosting  
Filtering  
Features  
Download  
Marketing  
Archives  
Weaver  
FAQ

GMANE

From: Mike Pall <mikelu-0911 <at> mike.de>  
Subject: **LuaJIT 2.0 intellectual property disclosure and research opportunities**  
Newsgroups: [gmane.comp.lang.lua.general](https://www.gmane.org/gmane.comp.lang.lua.general)  
Date: 2009-11-02 10:17:04 GMT (6 years, 27 weeks, 1 day, 22 hours and 41 minutes ago)



From the lua-@lists.lua.org mailing list

It has been brought to my attention that it might be advantageous for some parts of the research community and the open source community, that I make a public statement about the intellectual property (IP) contained in LuaJIT 2.0 and earlier versions:

I hereby declare any and all of my own inventions contained in LuaJIT to be in the public domain and up for free use by anyone without payment of any royalties whatsoever.

[Note that the source code itself is licensed under a permissive license and is not placed in the public domain. But this is an orthogonal issue.]

I cannot guarantee it to be free of third-party IP however. In fact nobody can. Writing software has become a minefield and any moderately complex piece of software is probably (unknowingly to the author) encumbered by hundreds of dubious patents. This especially applies to compilers. The current IP system is broken and software patents must be abolished. Ceterum censeo.

The usual form of disclosure is to write papers and publish them. I'm sorry, but I don't have the time for this right now. But I would consider publishing open source software as a form of disclosure.

In the interest of anyone doing research on virtual machines, compilers and interpreters, I've compiled a list of some of the new aspects to be found in LuaJIT 2.0. I do not claim all of them are original (I cannot possibly know all of the literature), but my research indicates that many of them are quite innovative.

This also presents some research opportunities for 3rd parties. I have little use for academic merits myself -- I'm more interested in coding than writing papers. Anyone is welcome to dig out any aspects, explore them in detail and publish them (giving due credit).

#### Design aspects of the VM:

- **NaN-tagging:** 64 bit tagged values are used for stack slots and table slots. Unboxed floating-point numbers (doubles) are overlaid with tagged object references. The latter can be distinguished from numbers via the use of special NaNs as tags. It's a remote descendant of pointer-tagging.

[The idea dates back to 2006, but I haven't disclosed it before 2008. Special NaNs have been used to overlay pointers before. Others have used it for tagging later on. The specific layout is of my own devising.]

- **Low-overhead call frames:** The linear, growable stack implicitly holds the frame structure. The tags for the base function of each call frame hold a linked structure of frames, using no extra space. Calls/returns are faster due to lower memory traffic. This also allows installing exception handlers at zero cost (it's a special bit pattern in the frame link).

Design of the IR (intermediate representation) used by the compiler:

I was already using NaN-tagging in 1991 to store registers kind & id in 64 bit numbers... [LD]

- **Linear, pointer-free IR:** The typed IR is SSA-based and highly orthogonal. An instruction takes up only 64 bits. It has up to two operands which are 16 bit references. It's implemented with a bidirectionally growable array. No trees, no pointers, no cry. Heavily optimized for minimal D-cache impact, too.
- **Skip-list chains:** The IR is threaded with segregated, per-opcode skip-list chains. The links are stored in a multi-purpose 16 bit field in the instruction. This facilitates low-overhead lookup for CSE, DSE and alias analysis. Back-linking enables short-cut searches (average overhead is less than 1 lookup). Incremental build-up is trivial. No hashes, no sets, no complex updates.
- **IR references:** Specially crafted IR references allow fast const vs. non-const decisions. The trace recorder uses type-tagged references (a form of caching) internally for low-overhead type-based dispatch.
- **High-level IR:** A single, uniform high-level IR is used across all stages of the compiler. This reduces overall complexity. Careful instruction design avoids any impact on low-level CSE opportunities. It also allows cheap and effective high-level semantic disambiguation for memory references.

Design of the compiler pipeline:

- **Rule-based FOLD engine:** The FOLD engine is primarily used for constant folding, algebraic simplifications and reassociation. Most traditional compilers have an evolutionary grown set of implicit rules, spread over thousands of hand-coded tiny conditionals.

The rule-based FOLD engine uses a declarative approach to combine the first and second level of lookup. It allows wildcard lookup with masked keys, too. A pre-processor generates a semi-perfect hash table for constant-time rule lookup. It's able to deal with thousands of rules in a uniform manner without performance degradation. A declarative approach is also much easier to maintain.

- **Unified stage dispatch:** The FOLD engine is the first stage in the compiler pipeline. Wildcard rules are used to dispatch specific instructions or instruction types (loads, stores, allocations etc.) to later optimization stages (load forwarding, DSE etc.). Unmatched instructions are passed on to CSE.

Unified stage dispatch facilitates modular and pluggable optimizations with only local knowledge. It's also faster than doing multiple dispatches in every stage.

#### Trace compiler:

- **NLF region-selection:** The trace heuristics use a natural-loop first (NLF) region-selection mechanism to come up with a close-to optimal set of (looping) root traces. Only special bytecode instructions trigger new root traces -- regular conditionals never do this. Root traces that leave the loop are aborted and retried later. This also gives outer loops a chance to inline inner loops with a low trip count.

NLF usually generates a superior set of root traces than the MRET/NET (next-executing tail) and LEI (last-executed iteration) region-selection mechanisms known from the literature.

- **Hashed profile counters:** Bytecode instructions to trigger the start of a hot trace use low-overhead hashed profiling counters. The profile is imprecise because collisions are ignored. The hash table is kept very small to reduce D-cache impact (only two

hot cache lines). Since NLF weeds out most false positives, this doesn't deteriorate hot trace detection.

[Neither using hashed profile counters, nor imprecise profiling, nor using profiling to detect hot loops is new. But the specific combination may be original.]

- **Code sinking via snapshots:** The VM must be in a consistent state when a trace exits. This means that all updates (stores) to the state (stack or objects) must track the original language semantics.

Naive trace compilers achieve this by forcing a full update of the state to memory before every exit. This causes many on-trace stores and seriously diminishes code quality.

A better approach is to sink these stores to compensation code, which is only executed if the trace exits are actually taken. A common solution is to emit actual code for these stores. But this causes code cache bloat and the information often needs to be stored redundantly, for linking of side traces.

Code sinking via snapshots allows sinking of arbitrary code without the overhead of the other approaches. A snapshot stores a consistent view of all updates to the state before an exit. If an exit is taken the on-trace machine state (registers and spill slots) and the snapshot can be used to restore the VM state.

State restoration using this data-driven approach is slow of course. But repeatedly taken side exits quickly trigger the generation of side traces. The snapshot is used to initialize the IR of the side trace with the necessary state using pseudo-loads. These can be optimized together with the remainder of the side trace. The pseudo-loads are unified with the machine state of the parent trace by the backend to enable zero-cost linking to side traces.

[Currently snapshots only allow store sinking of scalars. It's planned to extend this to allow arbitrary store and allocation sinking, which together with store forwarding would be a unique way to achieve scalar-replacement of aggregates.]

- **Sparse snapshots:** Taking a full snapshot of all state updates before every exit would need a considerable amount of storage. Since all scalar stores are sunk, it's feasible to reduce the snapshot density. The basic idea is that it doesn't matter which state is restored on a taken exit, as long as it's consistent.

This is a form of transactional state management. Every snapshot is a commit; a taken exit causes a rollback to the last commit. The on-trace state may advance beyond the last commit as long as this doesn't affect the possibility of a rollback. In practice this means that all on-trace updates to the state (non-scalar stores that are not sunk) need to force a new snapshot for the next exit.

Otherwise the trace recorder only generates a snapshot after control-flow constructs that are present in the source, too. Guards that have a low probability of being wrongly predicted do not cause snapshots (e.g. function dispatch). This further reduces the snapshot density. Sparse snapshots also improve on-trace code quality, because they reduce the live range of the results of intermediate computations. Scheduling decisions can be made over a longer stream of instructions, too.

[It's planned to switch to compressed snapshots. 2D-compression across snapshots may be able to remove even more redundancy.]

**Optimizations:**

- **Hash slot specialization:** Hash table lookup for constant keys is specialized to the predicted hash slot. This avoids a loop to follow the hash chain. Pseudocode:

```
HREFK:  if (hash[17].key != key) goto exit
HLOAD:  x = hash[17].value
-or-
HSTORE: hash[17].value = x
```

HREFK is shared by multiple HLOADs/HSTOREs and may be hoisted independently. The verification of the prediction (HREFK) is moved out of the dependency chain by a super-scalar CPU. This makes hash lookup as cheap as array lookup with minimal complexity.

It also avoids all the complications (cache invalidation, ordering constraints, shape mismatches) associated with hidden classes (V8) or shape inference/property caching (TraceMonkey).

- **Code hoisting via unrolling and copy-substitution (LOOP):** Traditional loop-invariant code motion (LICM) is mostly useless for the IR resulting from dynamic languages. The IR has many guards and most subsequent instructions are control-dependent on them. The first non-hoistable guard would effectively prevent hoisting of all subsequent instructions.

The LOOP pass does synthetic unrolling of the recorded IR, combining copy-substitution with redundancy elimination to achieve code hoisting. The unrolled and copy-substituted instructions are simply fed back into the compiler pipeline, which allows reuse of all optimizations for redundancy elimination. Loop recurrences are detected on-the-fly and a minimized set of PHIs is generated.

- **Narrowing of numbers to integers:** Predictive narrowing is used for induction variables. Demand-driven narrowing is used for index expressions using a backpropagation algorithm.

This avoids the complexity associated with speculative, eager narrowing, which also causes excessive control-flow dependencies due to the many overflow checks. Selective narrowing is better at exploiting the combined bandwidth of the FP and integer units of the CPU and avoids clogging up the branch unit.

**Register allocation:**

- **Blended cost-model for R-LSRA:** The reverse-linear-scan register allocator uses a blended cost model for its spill decisions. This takes into account multiple factors (e.g. PHI weight) and benefits from the special layout of IR references (constants before invariant instructions, before variant instructions).
- **Register hints:** The register allocation heuristics take into account register hints, e.g. for loop recurrences or calling conventions. This is very cheap to implement, but improves the allocation decisions considerably. It reduces register shuffling and prevents unnecessary spills.
- **x86-specific improvements:** Special heuristics for move vs. rename produce close to optimal code for two-operand machine code instructions.

Fusion of memory operands into instructions is required to generate high-quality x86 code. Late fusion in the backend allows better, local decisions, based on actual register pressure, rather than estimates of prior stages.

		<p>Ok, that's it! Sorry for the length of this posting, but I hope it was at least informative to someone out there.</p>
--	--	--

		<p>--Mike</p>
--	--	---------------