Optimizing Lua using run-time type specialization

Michael Schröder

Bachelor's Thesis, March 2012

Like other dynamically typed languages, Lua spends a significant amount of execution time on type checks. Yet most programs, even if they are written in a dynamic language, are actually overwhelmingly monomorphically typed. To remove this unnecessary type-checking overhead, we implement a portable optimization scheme that rewrites virtual machine instructions at run-time based on the types of their operands. While not consistent across all platforms, we achieve average speed-ups of 1.2x on Intel, with a threaded variant of our VM showing improvements in the 1.5x to 2.4x range.

1 *Introduction*

> Lua is a powerful, fast, lightweight, embeddable scripting language. [It] combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

`http://lua.org/about.html`

Like the language it interprets, the Lua virtual machine is small and clean, implemented in just under 15 KLoC. This makes it an excellent playing ground for interpreter optimization techniques. But the Lua VM is already pretty fast, compared to VMs of similar languages. Can we make it even faster?[1]

[1] Note that we will focus on *purely portable* optimizations, as is in the spirit of Lua. While just-in-time compilers can achieve speed-ups of multiple orders of magnitude compared to pure interpretation, they obviously trade speed for portability. Even if a platform is technically supported by a JIT compiler, there may be political or security-related reasons prohibiting the execution of self-modifying code (e.g. sandboxing on mobile devices). And as they are usually highly complex pieces of software, maintaining and extending JIT compilers is a decidedly non-trivial task. Besides, there already exists a JIT compiler for Lua, called LuaJIT (`http://luajit.org`), which incidentally is considered to be one of the fastest dynamic language implementations.

Usually, the first thing one attacks when trying to optimize an interpreter is its dispatch loop. The high number of branch mispredictions caused by naive dispatch implementations can have a major impact on performance [EG03b, EG03a]. However, this is only true for so called *low abstraction level interpreters*. The situation is quite different when looking at interpreters for *high-level* dynamic languages, such as Lua or Python. As Brunthaler has shown [Bru09], high abstraction level interpreters actually spend only a negligible amount of their total run-time doing instruction dispatch, because the amount of work necessary to execute each operation is comparatively high.

The obvious optimization approach, therefore, is to the reduce the amount of work per operation. In any dynamically typed language, a significant amount of time is spent on type checks. When the types of values are only knowable at runtime, even a seemingly simple operation like multiplying two numbers takes up quite a lot of CPU cycles. Add operator overloading into the mix (as is possible in Lua via the mechanism of metatables) and things suddenly become rather expensive.

Yet even though the possibility of dynamic typing exists, *most "dynamically typed" programs actually aren't*. In the overwhelming majority of cases, variables will stay monomorphic throughout the lifetime of the program.[2] This assumption provides the basis of our optimization scheme.

[2] An empirical study on a large number of programs written in the dynamically typed Icon language revealed a type consistency of 60% to 100%, with an average of about 80% [WG96].

2 *Run-time type specialization*

The idea is this: when an instruction is dispatched for the first time, it is *specialized* according to the types of its operands, i.e. its bytecode is rewritten so that the type knowledge is now inherent in the opcode of the instruction, eliminating the need for type checks when executing the operation.[3] To ensure that this is safe, we have to *guard* (that is, add type checks) to any instruction that could change the type of one of the operands of the instruction we just specialized.

[3] This is somewhat similar to the *quickening* technique used by the JVM and utilized by Brunthaler to implement inline caching and other optimizations for Python [Bru10a, Bru10b, Bru11].

Should one of the guards fail, we will *despecialize* all instructions that are depending on that guard. For simplicity's as well as performance's sake, we adopt a very black-and-white view of the world: types are either monomorphic (in the vast majority of cases) or highly polymorphic, meaning that we will despecialize at the first sign of trouble and never respecialize again.

Put another way: we are removing type checks from *loads* of values and adding them to *stores* of values. We hope that in the end this will eliminate more type checks than it introduces. [4]

[4] To see how this might work, consider that not every guarded instruction really needs to perform a type check, as the result type of an operation is often dependent on the type of its input. This will be demonstrated in greater detail in section 4.

3 *Prerequisites*

3.1 *Bytecode*

The Lua VM is a register machine, and a such its bytecode is actually more of a *word*code: each instruction is exactly 32 bits long and includes an opcode and up to three operands. Operands are most commonly registers (viz. indexes into the global Lua stack, offset by the base of the function) or constants (indexes into the function's array of constant values). Depending on the operation, the bytecode is internally partitioned in one of three ways:

31	22	13	5 0
B	C	A	OP
(s)Bx		A	OP
Ax			OP

Figure 1: Bytecode layout in the vanilla VM

With 6 bits available for the opcode, it is clear that there can only be 64 different instructions. 40 of those are actually used by the vanilla VM. This does not leave us with nearly enough space to add all the specialized and guarded instruction variants we need, which will be just shy of 200.

One possible solution to this problem is a variable-sized bytecode, where we can have as many opcode bits as we need and fetch additional operands on demand. This would be a major change to the VM however, and has its own complex performance implications (cf. [OKN02]).

Instead, we keep it simple: using the same fixed-size 32 bit instruction format, we only the change the internal partitioning, to four fields of one byte each:

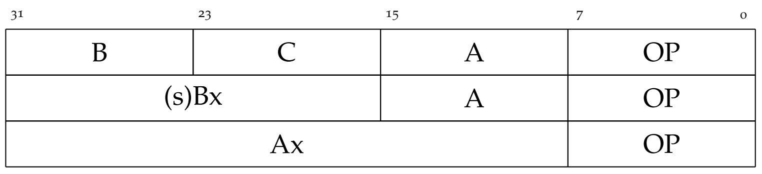


Figure 2: Bytecode layout in the special VM

An 8 bit opcode field gives us enough space for all the instructions we need, with room to spare. Alas, there is a tradeoff: by reducing the size of operands B and C, we impose some additional limitations on our modified VM:

	vanilla	special
possible opcodes	64	256
stack slots per function	250	120
constants per function	∼67 million	∼16.7 million
maximum jump offset	±131,071	±32,767

Table 1: Limits due to bytecode layout

None of these should have any real-world impact, however. Halving the maximum number of stack slots, for example, while effectively halving the maximum number of local variables that can be declared in a single function, still leaves us with a possible 120 named variables *per function* (including function arguments and minus a few registers needed to hold temporary results). This is a limit that hopefully no sane programmer would ever come close to.

3.2 *Register scope information*

The scope of a register *at a given point in time* (i.e. at some location of the program counter), is the range of instructions within which that register contains only a single semantic entity. There are two kinds of those entities:

Local variables All named variables, including function arguments. The scope of a local variable ranges from the first use of that variable to the last. Note that within this range there can be multiple stores and loads to and from the register.

Temporary variables These arise when the result of one operation is immediately used as input to another operation. Every temporary scope begins with a register store and ends when the register is first loaded from again.

It should be clear that within a function a single register can have multiple local and temporary scopes (because it can contain different semantic entities at different points in time), but that those scopes cannot overlap.[5] Gathering scope information for all registers of a function can be done at compile time and is possible with only a relatively small amount of changes to the bytecode compiler.

[5] Altough the borders of two scopes can, and quite often do, fall "within" the same instruction, e.g. `ADD 3 0 3` could end one temporary scope of register 3 and begin another one.

Note that *each register has to be stored to at least once within a scope before it is loaded from for the first time within that same scope.* In other words: any time an instruction that loads a variable is executed, an instruction that has stored that same variable *must* have been executed recently. This is what allows us to safely transfer type checks from loads to stores.

There is a small problem, though: this guarantee does not entirely hold in the case of function arguments and return values of calls. From [IdFC05]:

> For function calls, Lua uses a kind of *register window*. It evaluates the call arguments in successive registers, starting with the first unused register. When it performs the call, those registers become part of the activation record of the called function, which therefore can access its parameters as regular local variables. When this function returns, those registers are put back into the activation record of the caller.

Function arguments and return values are stored to their registers *outside* of the function they are used in, which might even be entirely outside the Lua environment. Register scopes cannot extend beyond function boundaries, which means these stores would be "invisible" to us, and there would be no way to guard them.

There is a simple solution, however: we introduce a pseudo-store instruction, which can be thought of as a "type change checkpoint". By default, this instruction does not perform any operation, it just acts as a sentinel to inform us that at this point in the execution the contents of a certain register might have changed. We name this instruction `CHKTYPE` and it takes exactly one argument, which is the register in question. `CHKTYPE` instructions are issued at the very beginning of a function, one for each function argument, and after `CALL` and `TFORCALL` instructions, one for each return value.

`CHKTYPE` instructions will also be issued after `VARARG` instructions. The `VARARG` operation stores multiple registers at once, and having one `CHKTYPE` for each store greatly simplifies specialization by allowing us to guard each register individually.

4 *Motivational example*

4.1 *Specialization*

The specialization process itself is actually rather simple, and is best explained with an example. We're going to specialize the small Lua function seen in listing 1.

Listing 1: A small Lua function

```
1 function f(a,n)
2   for i=1,n do
3     a[#a+1] = io.read()
4   end
5 end
```

The function reads `n` number of lines from the standard input and appends them to the given table `a`. Tables are associative arrays and provide the sole data structuring mechanism in Lua. Here, the table is used like a normal array. The length operator (`#`) seen on line 3 returns the number of elements in the table.

Given this function, the compiler of our modified VM will produce the bytecode seen in listing 2. It differs from bytecode produced by the vanilla VM in the addition of `CHKTYPE` instructions (at the beginning of the function and after the `CALL` on line 12), and of course in the fact that most instructions are of the *not-yet-specialized* variety, as indicated by the question mark on the right side of the operation name.

Listing 2: Bytecode before specialization

```
1       CHKTYPE       0
2       CHKTYPE       1
3       LOADK    num  2 -1
4       MOVE      ?   3  1
5       LOADK    num  4 -1
6       FORPREP   ?   7
7       LEN       ?   6  0
8       ADD       ?   6  6 -1
9       GETTABUP  ?   7  0 -2
10      GETTABLE  ?   7  7 -3
11      CALL          7  1  2
12      CHKTYPE       7
13      SETTABLE  ?   0  6  7
14      FORLOOP       2 -8
15      RETURN        0  1
```

Note that the two `LOADK` instructions are already specialized, since the types of their constant operands are known at compile time and are guaranteed not to change.

For the purposes of our example, we are skipping over the first couple of specializations, and begin with the program counter at line 7, right before `LEN ?` is executed (see listing 3). While the details differ from operation to operation, the basic procedure is always the same:

Listing 3: Before specialization of `LEN`

```
1       CHKTYPE       0
2   num CHKTYPE       1
3       LOADK    num  2 -1
4       MOVE     num  3  1
5       LOADK    num  4 -1
6       FORPREP  num  7
7       LEN       ?   6  0
8       ADD       ?   6  6 -1
9       GETTABUP  ?   7  0 -2
10      GETTABLE  ?   7  7 -3
11      CALL          7  1  2
12      CHKTYPE       7
13      SETTABLE  ?   0  6  7
14      FORLOOP       2 -8
15      RETURN        0  1
```

1. Examine the instruction's operands and determine their types at this point in time and if they are suitable for specialization.
2. For each operand, guard its register so that we can safely specialize. This means looking up the scope information for that register and then examining every single instruction within that scope, with the goal of finding those instructions that *store* to the register as part of their operation. How and if we add a guard depends on the instruction in question:
   a. If the instruction already guarantees the return type we want, then no guard is necessary. This is obviously the best case, as it completely eliminates the type check.
   b. If the instruction does *not* already have a guaranteed return type, then guarding it *is* necessary. This effectively transfers the type check.
   c. If instruction guarantees a return type but it is not the one we want, then we have hit upon a polymorphic type and abort the specialization process. Any other guards we might have added as part of this procedure up to now are removed again and the instruction we wanted to specialize is despecialized.
3. If adding all the necessary guards was successful, specializing the instruction is now safe. There is only one additional caveat: if the instruction we want to specialize is itself guarded, we need to reconcile this guard with the return type after specialization. If the new return type is the same as the guard, we can simply remove the guard and eliminate the type check completely.
   If the guard and the new return type clash, we immediately start the despecialization process on the result register, but our specialized instruction will stay specialized.
4. Re-dispatch the now specialized instruction.

In our example, the single operand of `LEN` on line 7 is register `0`, containing the local variable `a`, which is a table. Examining the scope of the operand, we only find one store and it is the pseudo-store of the `CHKTYPE` instruction on line 1. As per case b described above, we add a guard to `CHKTYPE` so that it becomes `tab CHKTYPE`. It is now safe for us to specialize `LEN ?` to `LEN tab` (see listing 4).

Listing 4: After specialization of `LEN`

```
1   tab CHKTYPE       0
2   num CHKTYPE       1
3       LOADK    num  2 -1
4       MOVE     num  3  1
5       LOADK    num  4 -1
6       FORPREP  num  7
7       LEN      tab  6  0
8       ADD       ?   6  6 -1
9       GETTABUP  ?   7  0 -2
10      GETTABLE  ?   7  7 -3
11      CALL          7  1  2
12      CHKTYPE       7
13      SETTABLE  ?   0  6  7
14      FORLOOP       2 -8
15      RETURN        0  1
```

Continuing in this vein, we now specialize `ADD ?`. This time there are two operands to check and guard. One of those is a constant, so no guards are necessary, and the other one is the temporary variable in register `6`, whose scope ranges from the result of `LEN tab` to the the input of `ADD ?`. Here we have the optimal situation were it is not necessary to add a guard, since the output of `LEN tab` is guaranteed to be a number. It is safe to specialize `ADD ?` to `ADD num` (see listing 5).

Listing 5: After specialization of `ADD`

```
1   tab CHKTYPE       0
2   num CHKTYPE       1
3       LOADK    num  2 -1
4       MOVE     num  3  1
5       LOADK    num  4 -1
6       FORPREP  num  7
7       LEN      tab  6  0
8       ADD      num  6  6 -1
9       GETTABUP  ?   7  0 -2
10      GETTABLE  ?   7  7 -3
11      CALL          7  1  2
12      CHKTYPE       7
13      SETTABLE  ?   0  6  7
14      FORLOOP       2 -8
15      RETURN        0  1
```

The rest of the function is specialized in the same fashion. The final result can be seen in listing 6. Take note of how the type checks that would have been necessary for `SETTABLE`, `ADD` and `LEN` could be eliminated by reducing them to the single type check at `tab CHKTYPE` on line 1. All in all we had to add three new type checks (on lines 1, 2 and 9), but could remove about twelve type checks by specializing the seven instructions on lines 4, 6-10 and 13.

Listing 6: Fully specialized function

```
1   tab CHKTYPE       0
2   num CHKTYPE       1
3       LOADK    num  2 -1
4       MOVE     num  3  1
5       LOADK    num  4 -1
6       FORPREP  num  7
7       LEN      tab  6  0
8       ADD      num  6  6 -1
9   tab GETTABUP str  7  0 -2
10      GETTABLE str  7  7 -3
11      CALL          7  1  2
12      CHKTYPE       7
13      SETTABLE num  0  6  7
14      FORLOOP       2 -8
15      RETURN        0  1
```

4.2 *Despecialization*

Our example function could be specialized by making some assumptions about its future. One of those assumption is that it will always be called with a table as the first argument. What happens if this is not the case?

We know what *should* happen: the function should behave just as it would if it was never specialized. This might either mean that it should produce a runtime exception, because whatever object we gave it instead of a table does not support the same operations, or it might mean that the function should continue to append lines read from the standard input to `a`, but maybe with a different interpretation of "append", depending on however the new object overloads the standard table operations. In either case, what the function must under no circumstances do is crash of some segmentation fault because one of the specialized instructions could no longer rely on doing things without a safety net. So we have to despecialize the function before any of this can happen. Again, the basic procedure is always the same:

1. Look up the scope information for the register whose guard has failed and examine every single instruction within that scope to find those that *load* from the register as part of their operation. What happens next depends on the particulars of each instruction:

   a. If the instruction is not itself guarded, simply remove the instruction's specialization (e.g. rewrite `LEN tab` to `LEN`).

b. If the instruction *does* have itself a guard, remove the instruction's specialization but keep the guard. There is no need to mess with the specializations of other instructions as long as it is possible for guards to keep them safe.

c. If the instruction does not itself have a guard, but did guarantee its return type by virtue of its specialization, remove the specialization and despecialize that instruction's result register as well.[6]

2. Remove all remaining guards of the register within the scope.

[6] Care must be taken not to get stuck in an infinite despecialization loop by remembering which registers have already been visited.

In our example, the failing guard of `tab CHKTYPE` on line 1 will lead to the despecialization of `LEN tab`, which in turn despecializes `ADD num`, which in turn despecializes `SETTABLE num`. When the process is finished, our function will look like listing 7, and will safely work with whatever type `a` now has. Note that not the whole function was despecialized, just those parts relating to the polymorphic type.

Listing 7: After despecialization of `a`

```
1        CHKTYPE       0
2    num CHKTYPE       1
3        LOADK    num  2 -1
4        MOVE     num  3  1
5        LOADK    num  4 -1
6        FORPREP  num  7
7        LEN           6  0
8        ADD           6  6 -1
9    tab GETTABUP str  7  0 -2
10       GETTABLE str  7  7 -3
11       CALL          7  1  2
12       CHKTYPE       7
13       SETTABLE      0  6  7
14       FORLOOP       2 -8
15       RETURN        0  1
```

4.3 *Upvalues*

One thing that we have neglected to mention until now is how we cope with local variables that have been captured in a closure, also known as upvalues. Since Lua has full closure support, assigning to an upvalue immediately assigns to the captured local variable in the enclosing function. This means that we need to be able to despecialize within the original scope of this local variable (in the enclosing function) when changing the type of the corresponding upvalue (in the enclosed function). In essence, we need to be able to despecialize "through" upvalues, i.e. across function boundaries.

Our implementation does this and goes a step further by also allowing *specialization* to occur through upvalues, which is possible by propagating guards across the function hierarchy:[7]

[7] Since not all of the information needed for this was available in the vanilla VM, we had to modify the virtual machine a little further so that the scopes of upvalues and their relation to parallel upvalues and enclosing functions is now fully collected at compile time and accessible to us during specialization.

- When specializing an instruction that has an upvalue operand (`GETUPVAL`, `GETTABUP` or `SETTABUP`), we add guards to all uses of the corresponding local variable of that upvalue in the enclosing function.
- When adding guards to a local variable, we recursively add guards to all `SETUPVAL` instructions in enclosed functions that store that local via the upvalue.
- When the type check of a guarded `SETUPVAL` fails, it despecializes the local variable in the enclosing function.
- When despecializing a local variable, we recursively despecialize all `GETUPVAL`, `GETTABUP` or `SETTABUP` instructions in enclosed functions that load that local via an upvalue.

5 Experimental Evaluation

5.1 Methodology

The sixteen benchmarks in our test suite were taken from the following sources:

- The Computer Language Benchmarks Game, a set of micro-benchmarks commonly used when comparing scripting language implementations.

`http://shootout.alioth.debian.org`

- Several variations of the Richards benchmark, which simulates the task dispatcher of a simple operating system kernel. The different versions make use of different features of the Lua language, such as metatables or tail calls.

`http://lua-users.org/lists/lua-l/2011-04/msg00609.html`

For more information see [dQ09]

- SciMark, a popular suite of scientific and numerical benchmarks, ported to Lua by Mike Pall. We have split up the individual components and gave them fixed iteration counts so as not to get an auto-scaled score.

`http://luajit.org/performance.html`

Both the vanilla and the special VM were compiled with `-O2 -fomit-frame-pointer`.[8] We measured user CPU times using the built-in `time` shell command on a variety of different platforms. For greater accuracy, benchmark inputs were chosen to produce run times around the mid double-digits whenever possible. The best results of three consecutive runs were compared.

[8] Apart from `-O2` being the default setting in Lua's makefiles, we found the results obtained using higher optimization levels to be very erratic, showing small improvements in a few cases, but exhibiting worse or unchanged performance most of the time. We attribute this to more aggressive inlining, but the full chain of cause-and-effect eludes us.

5.2 Results

Based on other purely interpretative efforts to reduce type-checking overhead in virtual machines, we expected to see relative speed-ups of at least 30% [WMG10, Bru09]. The nearest we come to this is on the Intel platform, were we could achieve a 20% speed-up on average and a maximum speed-up of nearly 60%. Still, the results are somewhat underwhelming, especially when we look at the other platforms were performance was significantly worse, with not even a 10% average speed-up for two of those.

	min	avg	max
Intel	0.96	1.21	1.59
AMD	0.96	1.08	1.22
PowerPC	0.95	1.11	1.26
ARM	0.95	1.07	1.17

Table 2: Relative speed-ups on different platforms

For a more detailed breakdown, see Appendix A

What is additionally troubling, is that we cannot explain why one of the platforms has fared so much better than the other three. Our first guess was that maybe the other machines' instruction caches were getting thrashed due to the increased size of our modified VM's main dispatch loop. But as it turns out, our modifications have *decreased* the overall amount of instruction cache misses. Additionally, the Intel machine actually has the second smallest instruction cache amongst the four platforms.

5.3 Threading and choice of compiler

To be fully compatible with ANSI C, the Lua VM implements switch-based dispatch. We stated before that this is generally not ideal in terms of performance, but that the impact for a high level VM such as Lua should be relatively minor. This assumption turned out to be wrong: after implementing a very simple version of indirect threading,[9] we found all around performance gains on the order of an additional 20% on the Intel platform, before and after type specialization, compared to switch dispatch.[10] This goes contrary to [Bru09] and shows that, at least for *some* high abstraction level interpreters, dispatch overhead *can* play a significant part in overall run time.

We additionally found that choice of compiler should not be underestimated. When we re-ran all benchmarks on the Intel platform for each of the modified VMs using different compilers, even though `gcc` proved the fastest choice for the vanilla VM, the specialized version using threading was an average 10% faster when compiled with `clang`, with an outlying peak speed-up of 2.45x on the mandelbrot benchmark. This goes to show that truly portable optimizations are indeed very hard to achieve, as choosing the right compiler and settings for the target platform can seriously influence the success of other optimizations.

[9] `http://lua-users.org/lists/lua-l/2010-11/msg00436.html`

[10] This is especially interesting when compared to the approach taken by Williams, McCandless and Gregg [WMG10]. With the same goal of reducing type checks, they replaced the default Lua dispatch loop wholesale with a graph-based "dynamic intermediate representation", which models control flow and type changes along its edges. Specialized nodes allow for efficient execution of operations based on the type knowledge inherent in the structure of the graph, similar to our specialized instructions. Dispatch along this graph also reduces the number of branch mispredictions in a similar way to threading. They achieved speed-ups of 1.3x on average, with peaks at 2x, though at the cost of using significantly more memory.
While the dynamic graph allows for further, more advanced optimizations and can be used as the basis to implement a full-on JIT compiler, our simpler scheme achieves comparable performance when combined with threading.

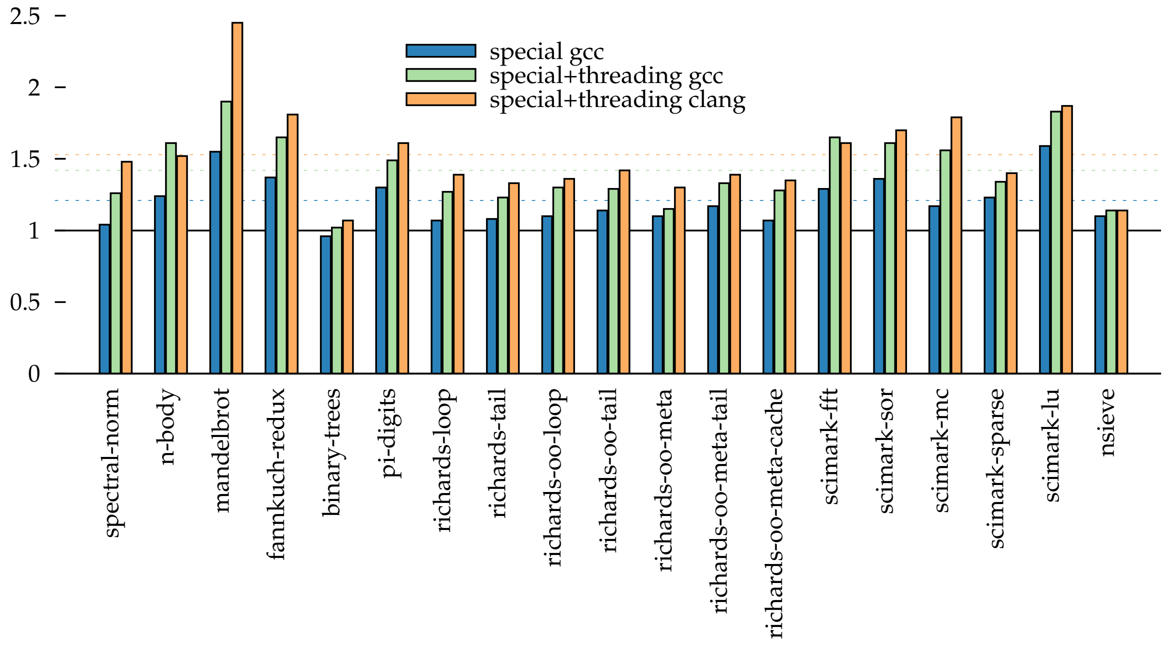


Figure 3: Relative speed-ups on the Intel platform for different optimizations and choices of compiler

References

[Bru09] Stefan Brunthaler. Optimizing high abstraction-level interpreters. In *Proceedings of the 26th Annual Workshop of the GI-FG 2.1.4 "Programmiersprachen und Rechenkonzepte" (Physikzentrum Bad Honnef, Germany, May 4-6, 2009), Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, Bericht Nr. 0915 (2009)*, pages 100–111, 2009.

[Bru10a] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages, Reno, Nevada, US, October 18, 2010 (DLS '10)*, pages 1–14, New York, NY, USA, 2010. ACM Press.

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25, 2010 (ECOOP '10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 429–451. Springer, 2010.

[Bru11] Stefan Brunthaler. *Purely Interpretative Optimizations*. PhD thesis, Vienna University of Technology, 2011.

[dQ09] Fabio Mascarenhas de Queiroz. *Optimized Compilation of a Dynamic Language to a Managed Runtime Environment*. PhD thesis, PUC-Rio, September 2009.

[EG03a] M Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5, 2003.

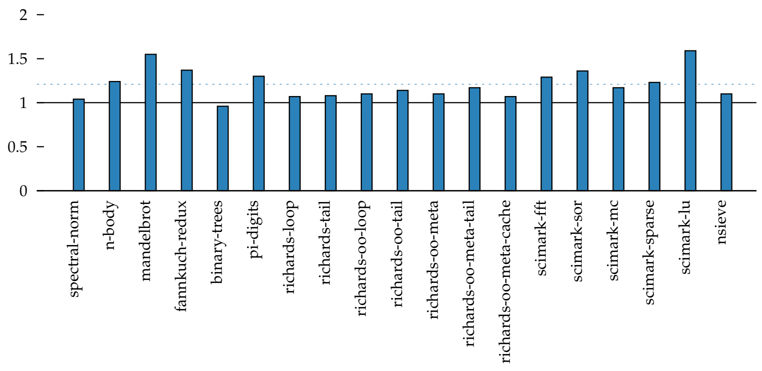
[IdFC05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

[OKN02] Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. Bytecode fetch optimization for a java interpreter. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 58–67, New York, NY, USA, 2002. ACM.

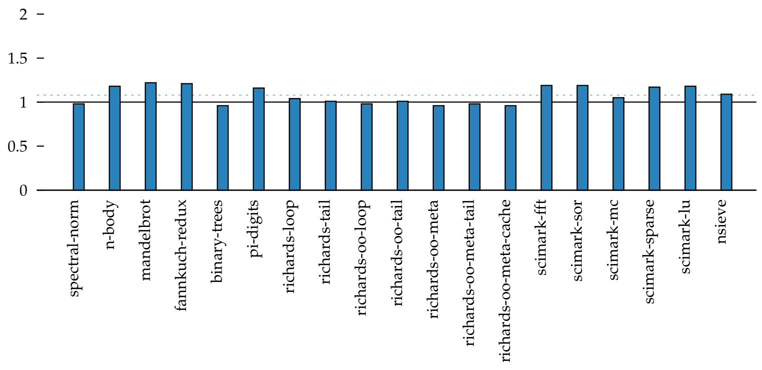
[WG96] Kenneth Walker and Ralph E. Griswold. Type inference in the icon programming language. Technical Report 93-32a, Department of Computer Science, University of Arizona, 1996.

[WMG10] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, CGO '10*, pages 278–287, New York, New York, USA, 2010. ACM Press.

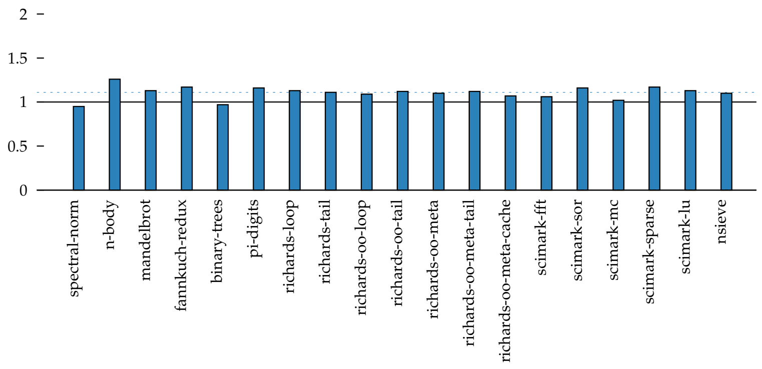
A Relative speed-ups on different platforms



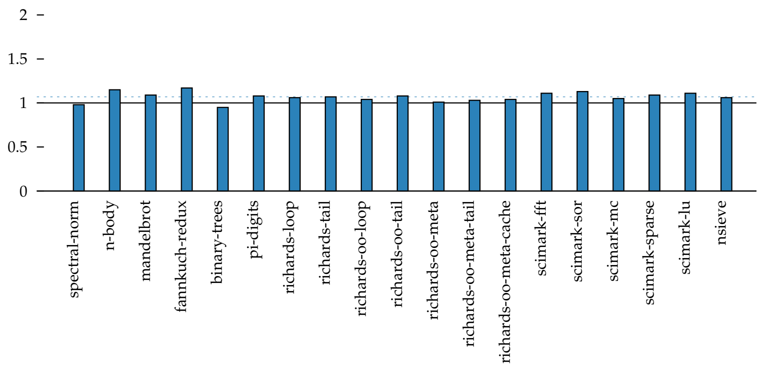
Intel Core 2 Duo with 2.13 GHz, running OSX 10.7.3 and `gcc` 4.2.1



AMD Athlon 64 X2 with 2.4 GHz, running Debian 4.1.1-21 and `gcc` 4.1.2



PowerPC 970 2 GHz, running Debian 4.4.5-8 and `gcc` 4.4.5



Feroceon 88FR131 1.2 GHz, an embedded ARM device running Debian 4.4.5-2 and `gcc` 4.4.5