4
Benchmarks

This chapter presents our suite of benchmarks and the results of those benchmarks on different variations of our Lua to CLR compiler and different implementations of the CLR. We analyze the performance impact of our changes to build a partial performance model of these implementations. We also benchmark our compilers against the Lua interpreter on x86, a Lua x86 JIT compiler, and a Microsoft-developed Python to CLR compiler. These benchmarks respectively serve to find out how well Lua can perform on the CLR relative to other Lua implementations, and how well our approach for compiling a dynamic language in the CLR works relative to a compiler using the Microsoft DLR.

Section 4.1 describes the programs we used in our benchmarks and the Lua operations and idioms that they exercise. Section 4.2 gives the results and analysis of our benchmarking of the different variations of our Lua to CLR compiler, and Section 4.3 compares compilers with other Lua implementations and with a Python compiler for the CLR. Appendix C has tables with running times for all of the benchmarks in this chapter.

4.1
Benchmark Programs

Our first suite of benchmarks is a suite of small (less than fifty lines of code) numerical benchmarks mostly taken from a suite of bencharks that compare several programming languages [Brent A. Fulgham and Isaac Gouy, 2009]. They are useful because implementations of these benchmarks are readily available for several programming languages, they have few dependencies on Lua's standard library, and they have no dependencies on the platform facilities. These benchmarks are naive, but useful for testing the impact of specific optimizations. Small benchmarks are useful for comparing performance of different implementations, and to guide programmers on how to tailor their programs to get the best performance out of a particular implementation [Gabriel, 1985].

The benchmarks are listed on Table 4.1, with a brief description of what they do and what they exercise (what are the main influences on their results).

Name	Description
binary-trees	allocation and traversal of binary trees, exercises small records, memory allocation and GC
fannkuch	array permutations with a small array, exercises array operations
fib-iter	fibonnaci function, iterative algorithm, exercises simple arithmetic and loops
fib-memo	fibonnaci function, recursive algorithm with memoization, exercises array operations on a variable-sized array and first-class functions
fib-rec	fibonnaci function, recursive algorithm, exercises recursion
mandelbrot	mandelbrot fractal, exercises floating point arithmetic and iteration
n-body	newtonian gravity simulation, exercises floating point arithmetic on records and arrays
n-sieve	sieve of Eratosthenes using an array, exercises array access
n-sieve-bits	sieve of Eratosthenes using bitfields, exercises floating point arithmetic and arrays
partial-sum	iterative summation of series, exercises floating point arithmetic and built-in functions with iteration
recursive	several recursive functions, exercises recursion
spectral-norm	spectral norm of an infinite matrix, exercises arithmetic involving several cooperating functions

Table 4.1: First benchmark suite

The overlapping coverage of these benchmarks of the first suite is on purpose; similar benchmarks should respond in a similar way to changes in the implementation of the compiler.

We also implemented a second suite of benchmarks. It is a set of variations on the Richards Benchmark [Richards, 1999], a medium-sized benchmark. The benchmark implements the kernel of a very simple operating system, with a task dispatcher, input/output devices, and worker tasks that communicate via message passing. The benchmark has implementations for several programming languages. Its output is deterministic, so it is trivial to verify that a particular implementation is correct; the simulation uses pseudorandom numbers, but the pseudorandom number generator is part of the benchmark.

We have six different implementations of the benchmark, with different ways of implementing the core task dispatcher; the implementations have around three hundred lines of Lua code each. For comparison, the C version of the benchmark has four hundred lines of code, and the Python version has about four hundred and fifty.

Our implementations of the Richards benchmarks are listed on Table 4.2.

Name	Description
richards	the closest to the C implementation, it uses an explicit state string and a sequence of ifs inside an endless loop as the core of the dispatcher; the state string simulates a bitfield
richards-tail	has a state machine using tail calls for the state transitions instead of an endless loop, with the code to implement each state factored out to a different function, but otherwise is identical to the previous one
richards-oo	embeds the dispatcher logic inside each task as several methods, and keeps the state local in each task object; state transition is via a trampoline: each task returns the next task to be processed
richards-oo-tail	the previous implementation using tail calls instead of the trampoline, each task calls the next one to be processed directly
richards-oo-meta	like richards-oo but uses a parent table to hold the methods, and a metatable associated with each task object that delegates method calls to this parent table
richards-oo-cache	also uses a parent table, but the metatable caches each method in the task object itself after the first use, to speed up subsequent lookups

Table 4.2: Second benchmark suite

4.2 Benchmarking the Variations

This section compares the different variations of our Lua compiler on different implementations of the CLR. We present a series of graphs of performance improvement (or worsening) relative to the base compiler described in Section 2.1. All graphs show a base 2 logarithm of the running time of the benchmark compiled by the base compiler divided by the running time of the benchmark compiled by each of the other compilers (e.g. -1 means the benchmark took twice as long, 2 means it took a fourth of the time). The other compilers are identified by the short names listed on Table 2.1. All of the tests are done on a computer with an AMD Phenom 8400 processor with 2Gb of RAM and executing in x86 (32-bit) mode.

The first CLR implementation we benchmarked is Microsoft .NET 3.5 SP1, the current version of the CLR released by Microsoft. Figure 4.1 shows the results of the first benchmark suite on this version of the CLR.

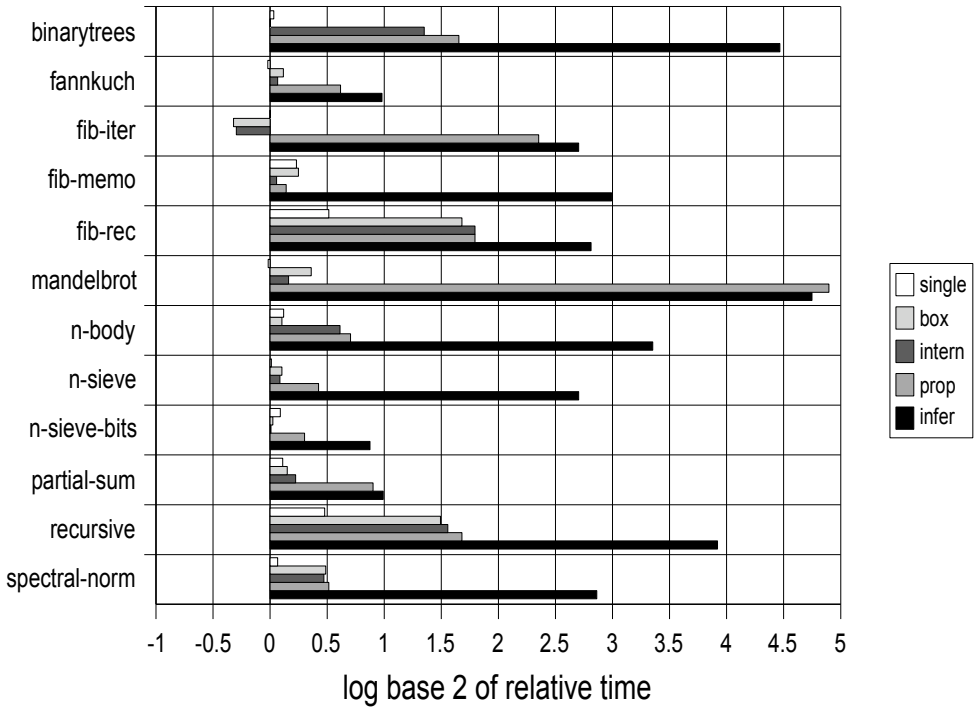

Figure 4.1: .NET 3.5 SP1 Comparison

The results show barely any improvement with the optimization of the function calls that only need to return a single value when we are using a structure to represent Lua values. Changing the representation to avoid structures and use boxing instead improves most benchmarks, specially the ones that make heavy use of function calls. The CLR JIT does not optimize code that uses structures as well as other code, so the performance gain in avoiding allocation of the unnecessary arrays gets lost in the noise [Morrison, 2008b].

Interning strings leads to a good improvement in the benchmarks that use records, confirming that this implementation of the CLR does not intern strings. Local type propagation shows a big improvement only in the benchmarks where the core of the benchmark is a numerical loop inside a single function, but for these benchmarks it is about as good as the full type inference. The biggest improvement comes from doing full type inference on the programs. The running time for the benchmarks of the first suite is dominated by boxing, type checking, and dispatch, and type inference is eliminating most of those.

The results for the second suite of benchmarks are on Figure 4.2. Neither the base compiler nor the single return value compiler could run the *richards-oo-tail*; they both blow the stack because the CLR JIT is not compiling the

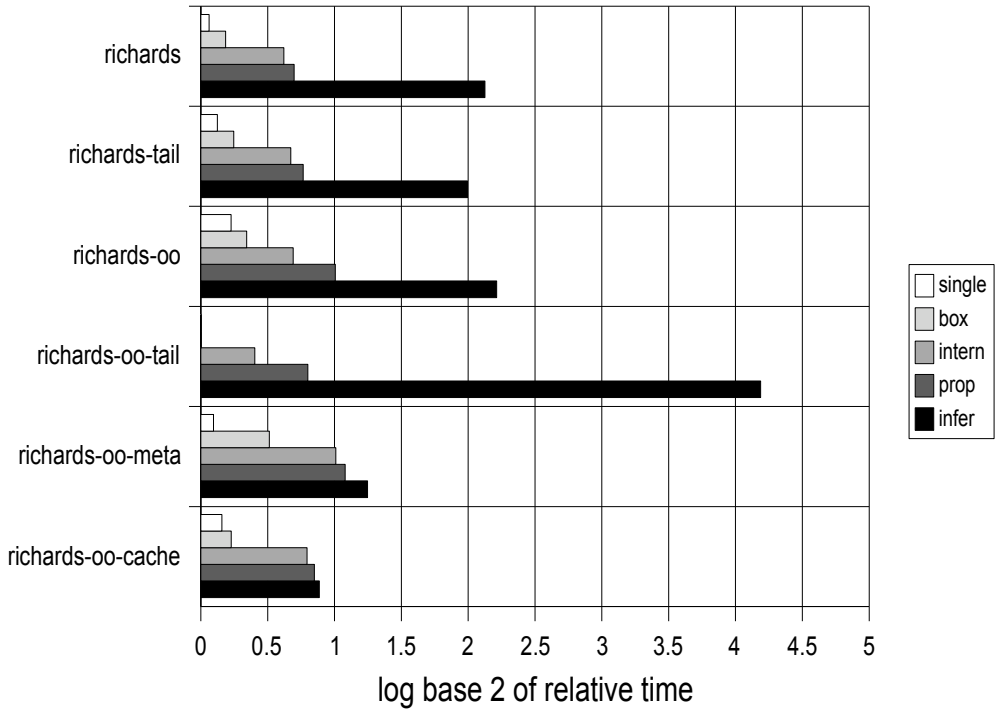

Figure 4.2: .NET 3.5 SP1 Comparison, Richards benchmarks

tail calls in them as actual tail calls, and they continue using stack space. The .NET JIT treats the tail call opcodes as just a suggestion, and ignores it in several situations [Broman, 2007]. Only the JIT on version 4.0 of .NET for the x64 platform will always honor the tail call opcodes [Richins, 2009]. The results for *richards-oo-tail* on Figure 4.2 are relative to the "box" compiler.

All of the benchmarks in this suite do a great number of string equality tests, so interning strings shows a good improvement for all of them. Local type propagation shows a modest improvement, but the biggest improvement again comes from doing full type inference. The big improvement in the *richards-oo-tail* benchmark for type inference relative to the improvement in *richards-oo* is due to another anomaly of tail calls in the .NET implementation of the CLR, where a tail call to a function with a different number of arguments than the current function interacts badly with the code that synchronizes with the thread running the garbage collector [Borde, 2005]; type inference is unifying all of the mutually recursive functions in *richards-oo-tail* to have the same signature, avoiding the problem and thus increasing the amount of improvement in relation to the other compilers.

The benchmarks that use metatables, *richards-oo-meta* and *richards-oo-cache*, show little improvement with type inference, as our type inference algorithm always infers the most general table type for tables that have a

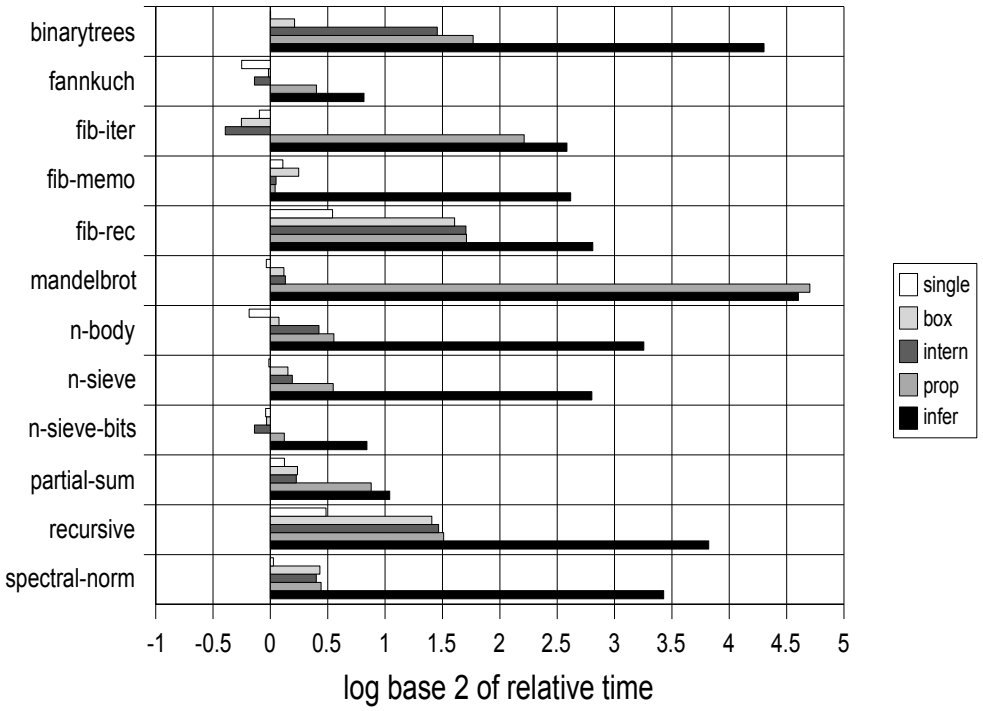

Figure 4.3: .NET 4.0 Beta 1 Comparison

metatable attached. This ends up spreading to functions used as methods.

We also ran our benchmarks on Microsoft .NET 4.0 Beta 1, the current beta of the next version of Microsoft's implementation of the CLR. The results are on Figure 4.3 and Figure 4.4. They are about the same as the results for .NET 3.5 SP1, showing that the behavior of the JIT compiler in both versions is similar. Microsoft .NET 4.0 Beta 1 has the same issues with tail calls that .NET 3.5 SP1 has, so our "base" and "single" compilers also cannot complete the *richards-oo-tail* benchmark.

Finally, we ran our benchmarks on Mono 2.4, an open-source implementation of the CLR. Figure 4.5 shows the results for the first suite of benchmarks. They are very different from the results of both Microsoft implementations, showing the different performance models of Mono and .NET, even though they both are implementations of the same managed runtime environment. Boxing in Mono performs much worse than using structures, and when using structures there is a good improvement in code that uses function calls when avoiding unnecessary array allocations. The best results are still obtained by doing full type inference.

Figure 4.6 shows the results for the suite of Richards benchmark on Mono 2.4. Arithmetic is not as critical for the benchmarks in this suite as in the benchmarks of the first suite, so the "box" compiler, which also optimizes

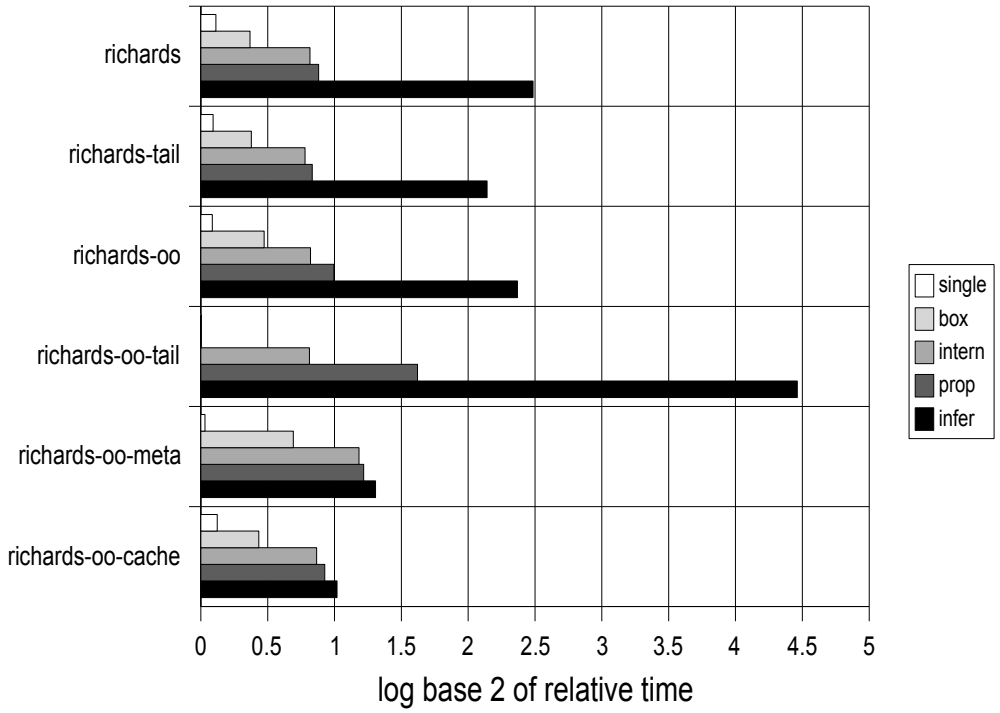

Figure 4.4: .NET 4.0 Beta 1 Comparison, Richards benchmarks

returning single values, shows an improvement in these benchmarks despite using boxed numbers. Mono also does not intern its strings, so we get a good improvement when doing that, relative to the "box" compiler. But none of the benchmarks that use tail calls ran on Mono without blowing the stack.

The differences between the Mono and .NET implementations are big enough to change the optimal compilation: for Mono the best approach is to combine type inference with a structure as a fallback uniform representation, while for .NET it is to combine type inference with a uniform representation that uses boxing, the one we actually implemented. Our previous work showed that the penalty for using structures was even greater in a previous version of the .NET CLR [Mascarenhas and Ierusalimschy, 2008]; the current version of the .NET CLR is just over one year old at the time of the writing of this dissertation, and is the first to have improved code generation for programs that use structures [Morrison, 2008b]. The performance characteristics vary not only between competing implementations of the same runtime, but also between different versions of the same implementation, so the best approach for building a compiler that targets a managed runtime environment can depend on a specific version of a specific implementation of the runtime.

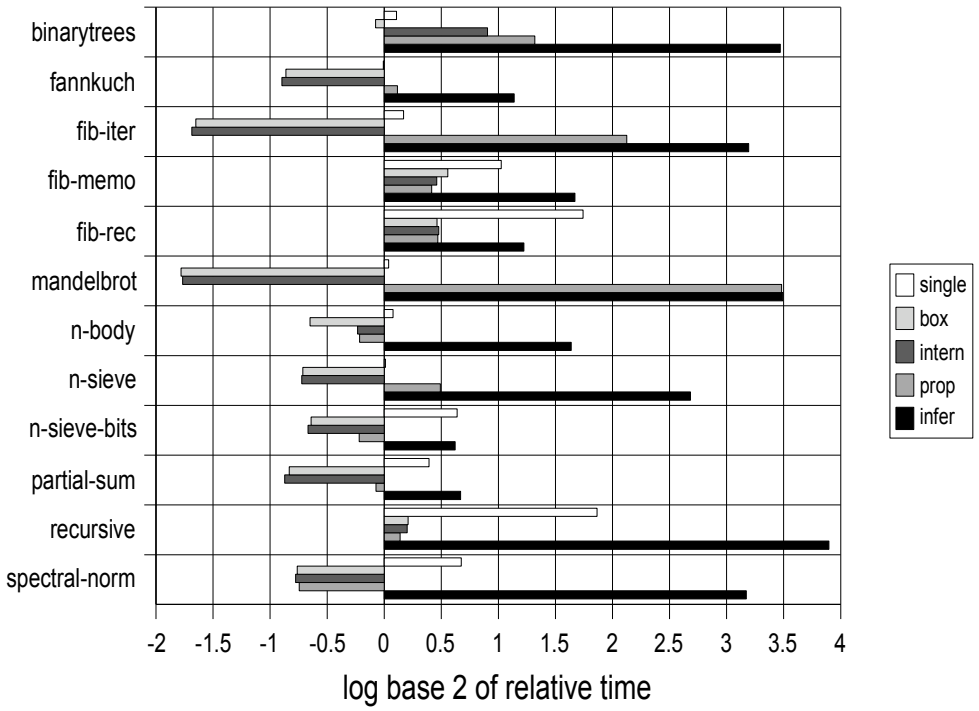

Figure 4.5: Mono 2.4 Comparison

4.3 Other Benchmarks

This section compares our Lua compilers first with other Lua implementations and then with IronPython 2.0, a Python compiler for the CLR that we already reviewed in Section 2.3. We first show the results of benchmarking our Lua compilers that were able to run all of the benchmarks in both suites ("box", "intern", "prop", and "infer") against version 5.1.4 of the Lua interpreter and LuaJIT 1.1.5, a JIT compiler for Lua 5.1. The graphs for this benchmark show the performance of each of our compilers, plus LuaJIT, relative to the Lua interpreter, also as the base 2 logarithm of the relative running times.

Figure 4.7 shows the results of the first benchmark suite. With type inference our last compiler is able to generate, with the help of the .NET JIT, code that performs better than LuaJIT for most benchmarks, and better than the Lua interpreter for all benchmarks. Local type propagation, in the three benchmarks where it was most useful, gets similar results. Our other two compilers still outperform the Lua interpreter in several benchmarks, but are worse than the interpreter in benchmarks that depend on floating point arithmetic, by a factor of two in some cases. The Lua interpreter is very efficient doing floating point computations, as it always works with unboxed numbers.

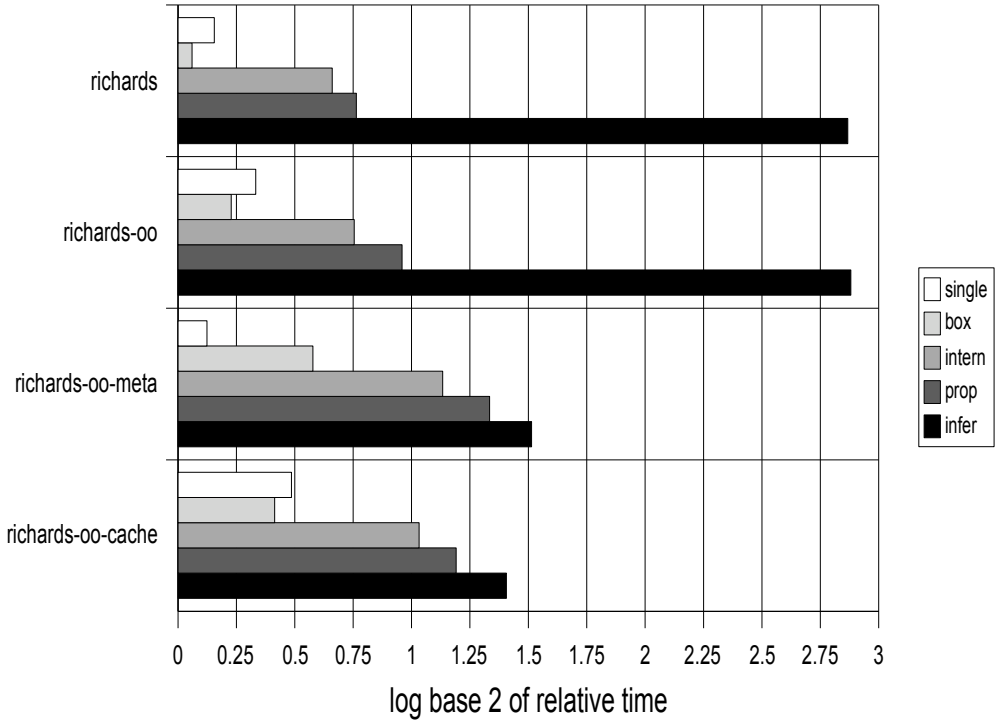

Figure 4.6: Mono 2.4 Comparison, Richards benchmarks

The results of the second benchmark suite are on Figure 4.8. The issue with tail calls in the .NET CLR is clearer to see here, as we are now comparing against the Lua interpreter. While the "box" compiler is about a factor of two slower than the Lua interpreter in most benchmarks, for the *richards-oo-tail* benchmark it is approximately ten times slower than the Lua interpreter. The gap narrows for the "intern" and "prop" compilers but is gone only for the last compiler, where this benchmark performs similarly to the *richards-oo* benchmark. This is consistent with the reason for the anomaly that we discussed in the previous section.

Combining the results of both benchmark suites, we have the benchmarks running in at most twice the time as the Lua interpreter, but usually running in a similar amount of time (except for the outlier, the *richards-oo-tail* benchmark). The performance is on par or exceeds the performance of a x86 Lua JIT compiler when our type inference algorithm is able to assign more precise types. These results support our assertion in Chapter 1 that it is possible to generate efficient code from a dynamically-typed source language to a managed runtime.

Our last benchmark compares our compilers with IronPython 2.0, a Python compiler for the CLR that we reviewed in Section 2.3. For this benchmark we added the *richards-oo* benchmark to the benchmarks of the

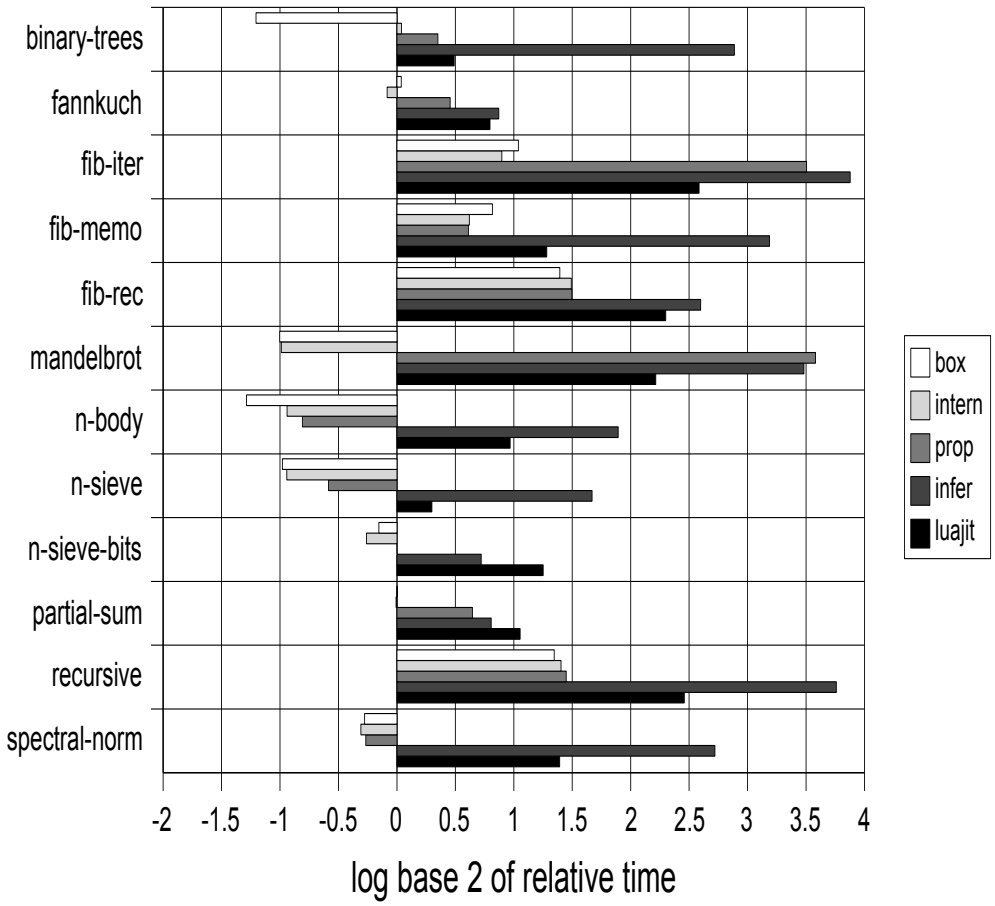

Figure 4.7: Comparison with Lua 5.1.4

first suite, as there is a single implementation of the Richards benchmark for Python. Figure 4.9 shows the performance relative to our "box" compiler, again by showing the base 2 logarithm of the relative running times ("ipy" is the IronPython compiler).

Python has separate array and dictionary types, while Lua arrays (in the absence of precise type inference) are an optimization of tables. This explains the better performance of IronPython in the *binarytrees*, *fib-memo* and *n-sieve* benchmarks, relative to our compilers that do not have type inference. Our compilers outperform IronPython in the other benchmarks even without type inference. With type inference we outperform IronPython in all of the benchmarks, in almost all of them by more than a factor of four, as the code IronPython generates is still doing runtime type checking and still boxing numbers.

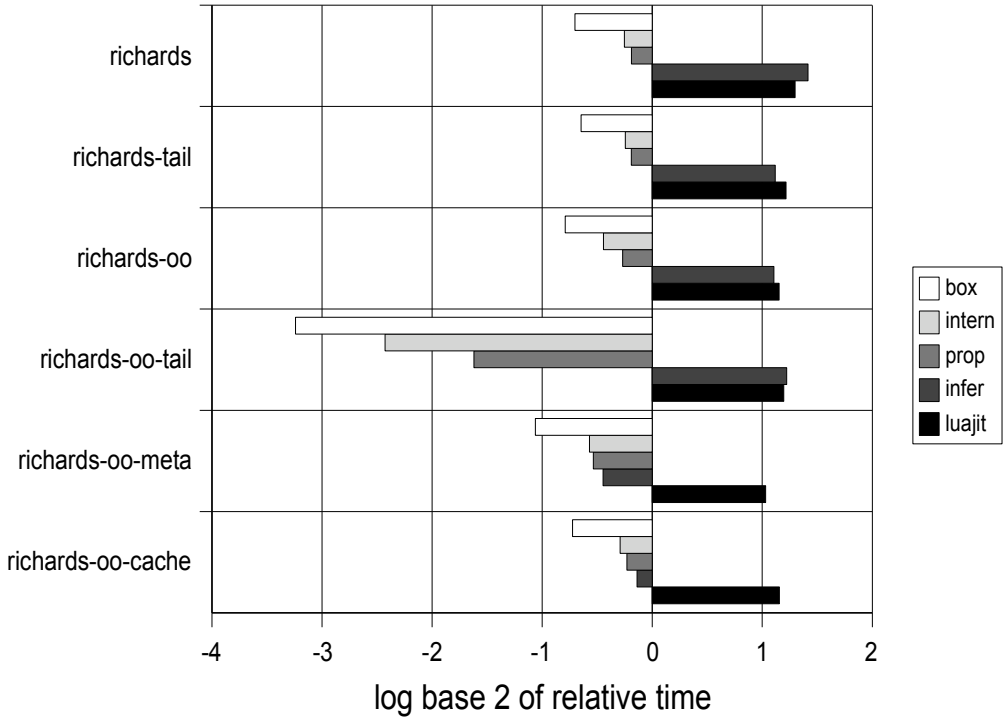

Figure 4.8: Comparison with Lua 5.1.4, Richards benchmarks

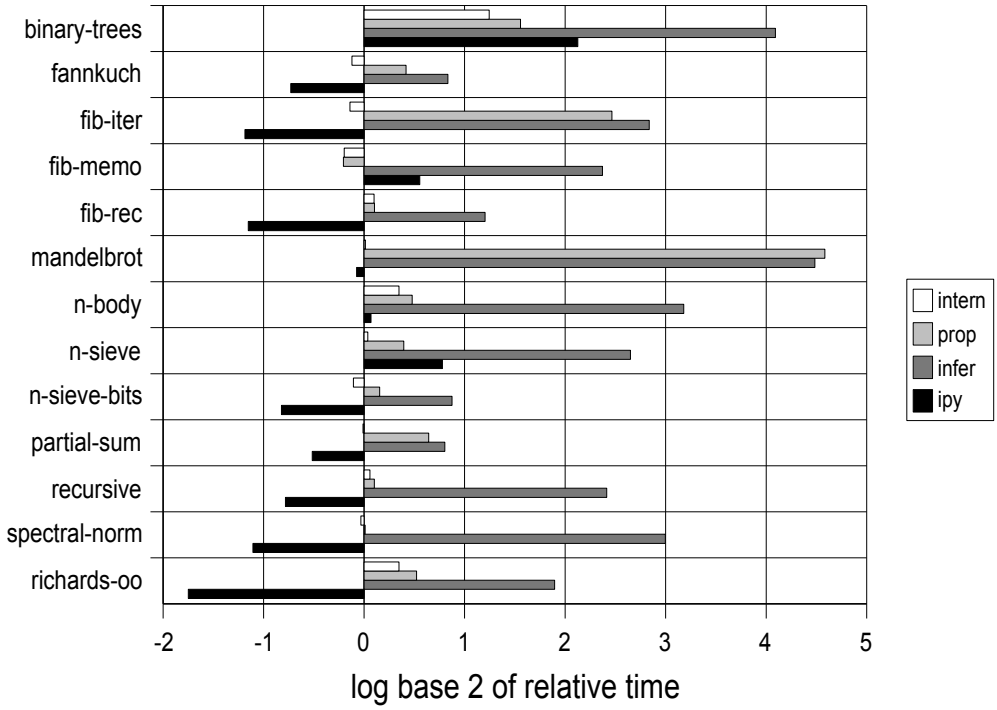

Figure 4.9: Comparison with IronPython