Supplementary Material for Rebuilding Racket on Chez Scheme (Experience Report)

MATTHEW FLATT, University of Utah, USA
CANER DERICI, Indiana University, USA
R. KENT DYBVIG, Cisco Systems, Inc., USA
ANDREW W. KEEP, Cisco Systems, Inc., USA
GUSTAVO E. MASSACCESI, Universidad de Buenos Aires, Argentina
SARAH SPALL, Indiana University, USA
SAM TOBIN-HOCHSTADT, Indiana University, USA
JON ZEPPIERI, independent researcher, USA

All benchmark measurements were performed on an Intel Core i7-2600 3.4GHz processor running 64-bit Linux. Except as specifically noted, we used Chez Scheme 9.5.3 commit 7df2fb2e77 at `github:cicso/ChezScheme`, modified as commit 6d05b70e86 at `github:racket/ChezScheme`, and Racket 7.3.0.3 as commit ff95f1860a at `github:racket/racket`.

1 TRADITIONAL SCHEME BENCHMARKS

The traditional Scheme benchmarks in figure 1 are based on a suite of small programs that have been widely used to compare Scheme implementations. The benchmark sources are in the `"common"` directory of the `racket-benchmarks` package in the Racket GitHub repository.

The results are in two groups, where the group starting with `scheme-c` uses mutable pairs, so they are run in Racket as `#lang r5rs` programs; for Racket CS we expect overhead due to the use of a record datatype for mutable pairs, instead of Chez Scheme's built-in pairs.

The groups are sorted by the ratio of times for Chez Scheme and the current Racket implementation. Note that the break-even point is near the end of the first group. The racket `collatz` benchmark turns out to mostly measure the performance of the built-in division operator for rational numbers, while `fft` and `nucleic` benefit from flonum unboxing.

2 SHOOTOUT BENCHMARKS

The benchmarks in figure 2 are based on a series of programs that have appeared over the years as part of the Computer Language Benchmarks Game to compare implementations of different languages.[1] The benchmark sources are in the `"shootout"` directory of the `racket-benchmarks` package in the Racket GitHub repository. We have only Racket implementations of these programs.

The groups are sorted by the ratio of times for Racket CS and the current Racket implementation. Results closer to the end of the table tend to rely more on Racket's hash tables, I/O, regular-expression matcher, thread scheduler, and flonum unboxing.

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Authors' addresses: Matthew Flatt, University of Utah, USA, mflatt@cs.utah.edu; Caner Derici, Indiana University, USA, cderici@indiana.edu; R. Kent Dybvig, Cisco Systems, Inc. USA, dyb@cisco.com; Andrew W. Keep, Cisco Systems, Inc. USA, akeep@cisco.com; Gustavo E. Massaccesi, Universidad de Buenos Aires, Argentina, gustavo@oma.org.ar; Sarah Spall, Indiana University, USA, sjspall@iu.edu; Sam Tobin-Hochstadt, Indiana University, USA, samth@cs.indiana.edu; Jon Zeppieri, independent researcher, USA, zeppieri@gmail.com.

Fig. 1. Traditional Scheme benchmarks. Shorter is better. CS = unmodifed Chez Scheme, CS′ = modified Chez Scheme, R/CS = Racket CS, R = current Racket implementation.

Fig. 2. Shootout benchmarks. Shorter is better. R/CS = Racket CS, R = current Racket implementation.

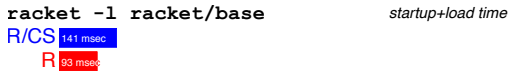
3 STARTUP TIMES

Startup for just the runtime system without any libraries:

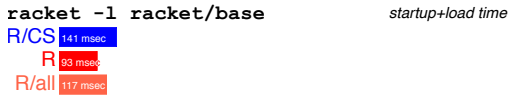
The Racket CS startup image has much more Scheme and Racket code that is dynamically loaded and linked, instead of loaded as a read-only code segment like the compiled C code that dominates

the current Racket implementation. We can build the current Racket implementation in a mode where its Racket-implemented macro expander is compiled to C code instead of bytecode, too, shown below as "R/cify." We can also compare to Racket v6, which had an expander that was written directly in C:

Loading the `racket/base` library:



Racket CS's machine code is bigger than current Racket's bytecode representation. Furthermore, the current Racket implementation is lazy about parsing some bytecode. We can disable lazy bytecode loading with the `-d` flag, shown as "R/all":

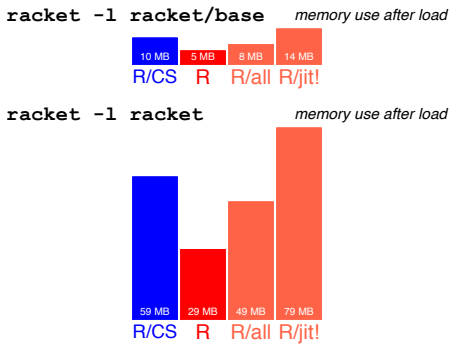


Loading the larger `racket` library, which is what the `racket` executable loads by default for interactive mode:

The measurements in this section were gathered by using `time` in a shell a few times and taking the median. The command was as shown, but using `racket -d` for the "R/all" lines.

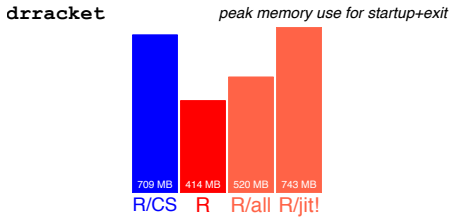
4 MEMORY USE

The following plots show memory use, including both code and data, after loading `racket/base` or `racket`, but subtracting memory use at the end of a run that loads no libraries (which reduces noise from different ways of counting code in the initial heap). The "R/jit!" line uses `-d` to load all bytecode eagerly, and it further forces that bytecode to be compiled to native code by the JIT compiler.



These results show that bytecode is more compact than machine code, as expected. Lazy parsing of bytecode also makes a substantial difference in memory use for the current Racket implementation. Racket's current machine code takes a similar amount of space as Chez Scheme machine code, but the JIT overhead and other factors make it even larger. (Bytecode is not retained after conversion to machine code by the JIT.)

On a different scale and measuring peak memory use instead of final memory use for DrRacket start up and exit:

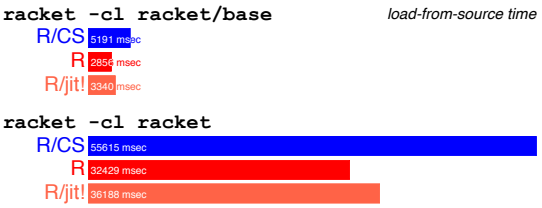


This result reflects that DrRacket's memory use is mostly the code that implements DrRacket, at least if you just start DrRacket and immediately exit.

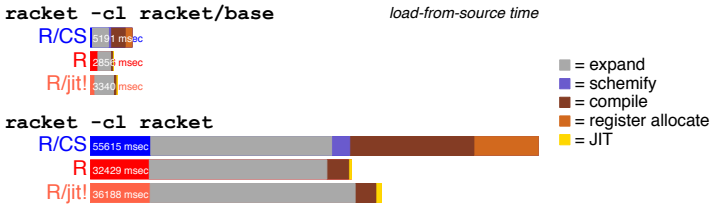
The measurements in this section were gathered by running `racket` starting with the arguments `-l racket/base`, `-l racket`, or `-l drracket`. The command further included `-W "debug@GC" -e '(collect-garbage)' -e '(collect-garbage)'` and recording the logged memory use before that second collection. For the "R" lines, the reported memory use includes the first number that is printed by logging in square brackets, which is the memory occupied by code outside of the garbage collector's directly managed space. For "R/all," the `-d` flag is used in addition, and for "R/jit!," the `PLT_EAGER_JIT` environment variable was set in addition to supplying `-d`. DrRacket's peak memory use was measured by waiting for the background expansion indicator to turn green for an empty program, and the result for `racket` add the last recent memory use reported for place 1.

5 EXPAND AND COMPILE TIMES

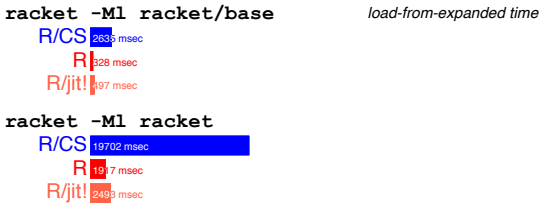
These plots compare compile times from source for the `racket/base` module (and all of its dependencies) and the `racket` module (and dependencies):



Compilation requires first macro-expanding source. Racket CS and current Racket use the same expander implementation. The following plots show how parts of the compile time can be attributed to specific subtasks:

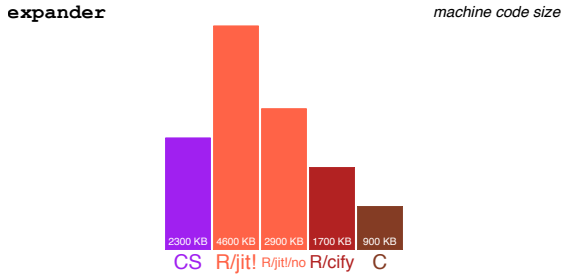


We can alternatively start with modules that are already expanded by the macro expander and just compile them:



We can make a relatively direct comparison of compile times between C and Racket, because the Racket macro expander was formerly written in C, and now it is written in Racket with essentially the same algorithms and architecture. The implementations are not so different in lines of code: 45 KLoC in C versus 28.5 KLoC in Racket. The following plot shows compile times for the expander's implementation:

To further check that we're comparing similar compilation tasks, we can check the size of the generated machine code. We can compile the Racket code to C code through a `cify` compiler. Below is a summary of machine-code sizes for the various compiled forms of the expander:



The current Racket implementation generates much more code from the same implementation, in part because it inlines functions aggressively and relies on the fact that only called code is normally translated to machine code; the "R/jit!/no" bar shows the code size when inlining is disabled.

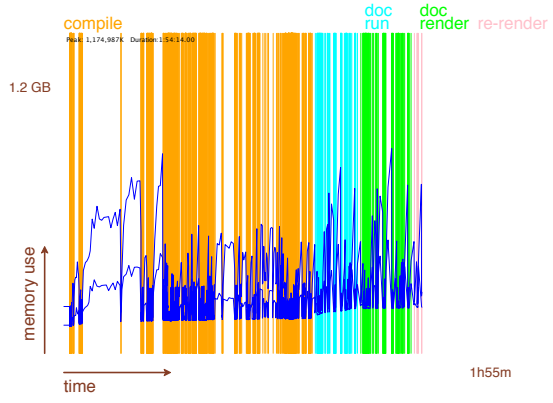
> The measurements in compile-time plots come from running the shown command (but with `racketcs` instead of `racket` for the "R/CS" lines) with the `PLT_EXPANDER_TIMES` and `PLT_LINKLET_TIMES` environment variables set. The overall time is as reported by `time` for user plus system time, and the divisions are extracted from the logging that is enabled by the environment variables.
> For measuring compile times on the expander itself, the Chez Scheme measurement is based on the build step that generates `"expander.so"`, the current-Racket measurement

is based on the build step that generates `"cstartup.inc"`, and the C measurement is based on subtracting the time to rebuild Racket version 6.12 versus version 7.2.0.3 when the `".o"` files in `"build/racket/gc2"` are deleted.
For measuring machine-code size, the expander's code size for Chez Scheme was computed by comparing the output of `object-counts` after loading all expander prerequsites to the result after the expander; to reduce the code that is just form the `library` wrapper, the expander was compiled as a program instead of as a library. The code size for Racket was determined by setting `PLT_EAGER_JIT` and `PLT_LINKLET_TIMES` and running `racket -d -n`, which causes the expander implemtation to be JITted and total bytes of code generated by the JIT to be reported. The "R/no-inline" variant was the same, but compiling the expander to bytecode with `compile-context-preservation-enabled` set to `#f`, which disables inlining. The "R/cify" code size was computed by taking the difference on sizes of the Racket shared library for a normal build and one with `--enable-cify`, after stripping the binaries with `strip -S`, then further subtracting the size of the expander's bytecode as it is embedded in the normal build's shared library. The "C" code size was similarly computed by subtracting the size of the Racket shared library for version 7.2.0.3 from the size for the 6.12 release, stipped and with the expander bytecode size subtracted.

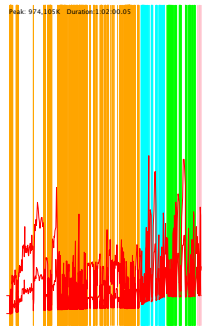
6 BUILD PROFILE

Building the Racket distribution from source involves compiling Racket code, running documentation to gather cross-reference information, rendering that documentation to HTML form, and the re-rendering some documentation to reach a fixed point. Plots in this section show memory use plotted against time for building the Racket distribution from source, all on the same scale.

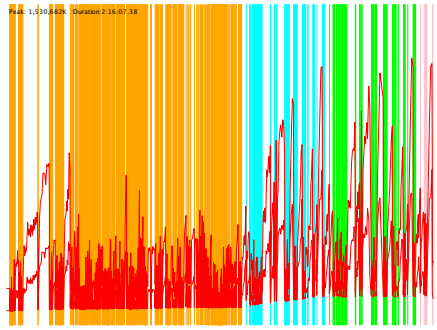
For **Racket CS**:



For the current **Racket** implementation:



To partly separate the cost of macro expansion and module loading from the cost of compilation after expansion, the following plots show build activity when using current **Racket** and making "compile" just mean "expand":



Given the result of the expand-only build as an input, we can then compile each fully expanded module to machine code. For **Racket CS**:

For the current **Racket** implementation:

The shortness of these last two plots illustrate that the overall time to build Racket from source is not so much from compile-time differences as other end-to-end performance effects related to loading and instantiating compile-time modules for macro expansion. We expect to be able to improve those effects without having to fundamentally change the approach to compilation in Racket CS.

These plots in this section were generated using the `"plt-build-plot"` package, which drives a build from source and plots the results. The build with “compile” as “expand” was created by using the `-M` flag, and then the finishing builds were measured by another run on the result.
We used Chez Scheme 9.5.3 modified as commit a48f3525d7 at `github:racket/ChezScheme` and Racket 7.3.0.4 as commit 0bffb7035d at `github:racket/racket`. This more recent version corrects a memory leak that was large enough to be visible in the plots.