`Core_bench`: micro-benchmarking for OCaml

Christopher S. Hardin and Roshan P. James

June 30, 2013

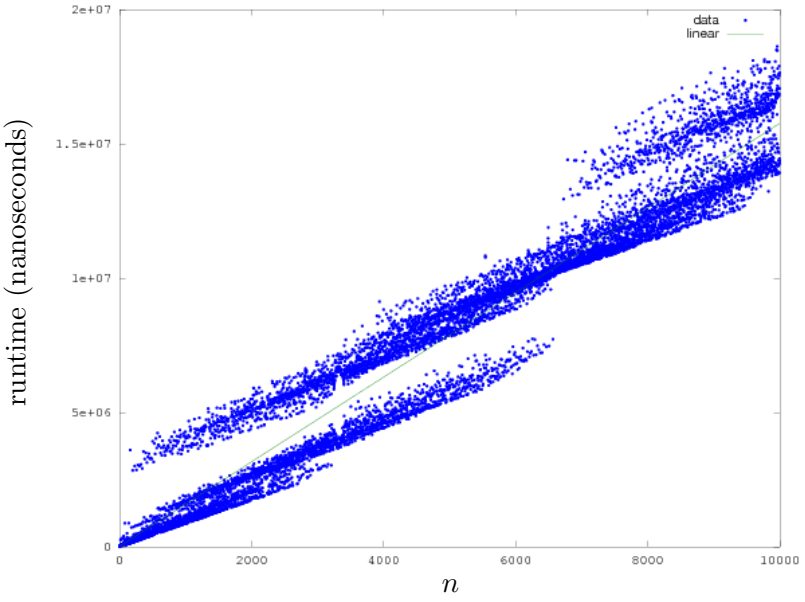
Abstract

We present `Core_bench`, an open source micro-benchmarking library for OCaml available through the OCaml package manager `opam` [1]. It is distinguished from other micro-benchmarking suites in its ability to account for garbage collection overheads when reporting the costs of functions.

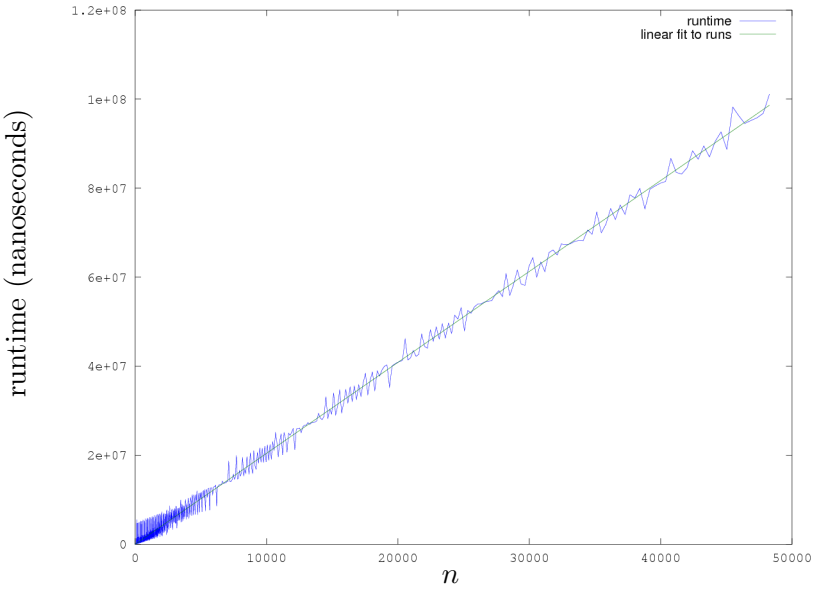
`Core_bench` was developed at Jane Street Capital, where we use OCaml to develop most of our software including performance-critical systems such as trading systems. Micro-benchmarking is an important tool in developing fast OCaml code because it makes execution costs hidden behind OCaml's high-level abstractions explicit. Developing an understanding of such low-level costs guides program design in performance-critical scenarios.

Micro-benchmarking is much harder than naive timing of functions because such timing is susceptible to the cost and resolution of the timing function and noise introduced by system load. Established micro-benchmarking libraries, such as Criterion developed for Haskell [2], tackle this by determining a number of runs n for a given function f such that the cost and resolution of the timing function are negligible compared to the cost of n runs of f. The library then collects several samples of n runs of f and perform some statistical analysis on the batch run times to determine outliers and confidence.

A drawback of that approach is that results can depend on the batch size n in non-obvious ways. For example, in the following graph, where runtime is plotted against batch size, one can see that a batch size of $n = 6700$ yields lower-variance measurements than a batch size of 5000, since the number of GCs per batch is more consistent in the former.

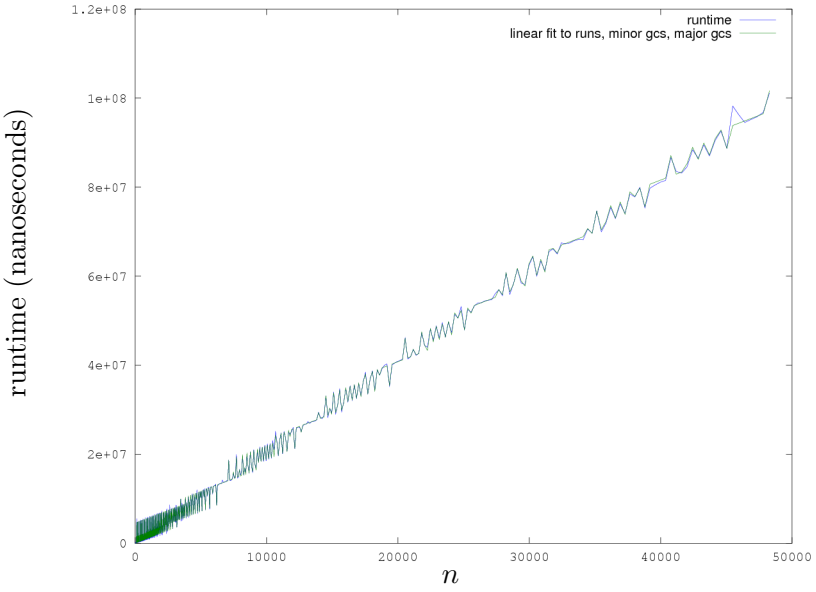


This reveals another reason why micro-benchmarking is complicated: execution times hide deferred GC costs, which are the source of the variance in the above graph. Rather than using a fixed batch size n, `Core_bench` iterates through many different n. Letting $t(n)$ denote the measured runtime of a batch of size n, `Core_bench` fits a line to the points $(n, t(n))$; the slope of that line is then the estimated execution time of f:



It is worth noting that the costs of the timing function (and any other such constant overhead) do not bias the results, since they only contribute to the y-intercept of the best-fit line, not its slope. `Core_bench` is given a time quota for benchmarking a function. This time quota is spent in sampling $t(n)$ for geometrically increasing n, such that we explore more of the x-axis and consequently get a more reliable fit. There are many statistical techniques available for estimating the error of this slope; `Core_bench` uses bootstrapping and also reports the coefficient of determination (R^2). If the reported numbers indicate low confidence, one can increase the time quota.

Separating nominal cost from GC cost. `Core_bench` attempts to decompose the cost of f into its nominal execution cost and the cost of GC. It does so currently, in a limited way, by reporting the regression slope for data samples that incur no major GCs. We are currently investigating a natural extension of our approach by using the number of minor collections, major collection and compactions in addition to n as predictors for the regression. This analysis also yields the costs per minor GC and major GC as regression results without any additional OCaml runtime instrumentation. Preliminary results are promising and suggest that these GC statistics account for almost all of the noise, as shown in the following plot of the multi-variable linear model over the sampled data:



Further work. `Core_bench` currently uses ordinary least squares for its regressions. This does not account for the fact that outliers in benchmarking are almost always on the positive side. One could replace ordinary least squares with a Bayesian method that incorporates our beliefs about the distribution of noise in measurements to get slightly more accurate results.

References

[1] OPAM, a package manager for OCaml. `http://opam.ocamlpro.com`.

[2] Bryan O'Sullivan. Criterion. `https://github.com/bos/criterion`.