The `nofib` Benchmark Suite of Haskell Programs

Will Partain
University of Glasgow

Abstract

This position paper describes the need for, make-up of, and "rules of the game" for a benchmark suite of Haskell programs. (It does not include results from running the suite.) Those of us working on the Glasgow Haskell compiler hope this suite will encourage sound, quantitative assessment of lazy functional programming systems. This version of this paper reflects the state of play at the initial pre-release of the suite.

1 Towards lazy functional benchmarking

1.1 History of benchmarking—functional

The quantitative measurement of systems for lazy functional programming is a near-scandalous subject. Dancing behind a thin veil of disclaimers, researchers in the field can still be found quoting "nfibs/sec" (or something equally egregious), as if this refers to anything remotely interesting.

The April, 1989, *Computer Journal* special issue on lazy functional programming is a not-too-dated self-portrait of the community that promotes computing in this way. It is one that non-specialists are likely to see. There are three papers under the heading "Efficiency of functional languages."

The Yale group, reporting on their ALFL compiler, cites results for the benchmarks `queens 8`, `nfib 20`, `tak`, `mm`, `deriv`, `tfib 100 40`, `qsort`, and `init`, noting that several are from the Gabriel suite of LISP benchmarks. They say that results from these tests "indicate that functional languages are indeed becoming competitive with conventional languages" [2, page 160].

Augustsson and Johnsson have a section about performance in their paper on the LML compiler [1]. They consider some of the usual suspects: `8queens`, `fib 20`, `prime`, and `kwic`, comparing against implementations of these algorithms in C, Edinburgh and New Jersey SML, and Miranda.[1] To their credit, the wondrous Chalmers hackers are somewhat apologetic, conceding that "measuring performance of the compiled code is very difficult..."

Finally, Wray and Fairbairn argue for programming techniques that make "essential use of non-strictness" and for an implementation (TIM) that makes these techniques inexpensive [10]. Though they delve into a substantial spreadsheet-like example program, they do not report any actual measurements. However, they astutely take issue with the usual toy benchmarks: "There was in the past a tendency for implementations to be judged on their performance for unusually strict benchmarks."

[1] Miranda is a trademark of Research Software Ltd.

1.2 History of benchmarking—imperative

Our imperative-programming colleagues are not far removed from our brutish benchmarking condition. Only a few years ago, "MIPS ratings," Dhrystones and friends were all the rage: marketeers bandied them about shamelessly, compiler writers tweaked their compilers to spot specific constructs in certain benchmarks, users were baffled, and no-one learned much that was worth knowing. The section on performance in Hennessy and Patterson's standard text on computer architecture is an admirable exposé of these shenanigans and is well worth reading [7].

Then, in 1988, enter the Standard Performance Evaluation Corporation (SPEC) benchmarking suite. The initial version included source code and sample inputs ("workloads") for four mostly-integer programs and six mostly-floating-point programs. These are all either real programs (e.g., the GNU C compiler) or the "kernel" of a real program (e.g., `matrix300`, floating-point matrix multiplication). Computer vendors have since put immense effort into improving their "SPECmarks," and this has delivered real benefit to the workstation user.

1.3 Towards lazy benchmarking

The SPEC suite is the most visible artifact of an important shift towards *system* benchmarking. A big reason for the shift lies in the benchmarked systems themselves. Fifteen years ago, a typical computer system—hardware and software—probably came from one manufacturer, sat in one room, and was a computing environment all on its own.

An excellent discussion about benchmarking from the self-contained-systems era is Gabriel and Masinter's paper about LISP systems [4]. "The proper role of benchmarking is to measure various dimensions of Lisp system performance and to order those systems along each of these dimensions" (page 136). A toy benchmark, of the type I have derided so far, can focus on one of these "dimensions," thus contributing to an overall picture.

Much early measurement work in functional programming was of this plot-along-dimensions style; however, the concern was usually to assess a particular implementation technique, not the system as a whole. For example, Hailpern, Huynh and Révész tried to compare systems that use strict versus lazy evaluation [5]. They went to considerable effort to factor out irrelevant details, hoping to end up with pristine data points along interesting dimensions. Hartel's effort to characterise the relative costs of fixed-combinator versus program-derived-combinator implementations was even more elaborate, using non-toy SASL programs [6].

So, can SPEC be seen as a culmination of good practice in benchmarking-by-characteristics? No! SPEC makes *no effort* to pinpoint systems along "interesting" dimensions, except for the very simplest—elapsed wall-clock time. An underlying premise of SPEC is that systems are sufficiently complicated that we probably won't even be able to pick out the interesting dimensions to measure, much less characterise benchmarks in terms of them. SPEC represents a shift to *lazy* benchmarking *of systems*.

Conte and Hwu's survey confirms that, at least in computer architecture, this shift towards "lazy, system-oriented benchmarking" is supported as a Good

Thing [3]. The trend can also be seen in some specialised areas of computing: the Perfect Benchmarks for supercomputers (Crays, etc., running FORTRAN programs) [8] and the Stanford benchmarks for parallel, shared-memory systems [9] are two examples.

2 Some serious benchmarks, `nofib`

We, the Glasgow Haskell compiler group, wish to (help) develop and promote a freely-available benchmark suite for lazy functional programming systems—called the `nofib` suite—consisting of:

1. Source code for "real" Haskell programs to compile and run;
2. Sample inputs (workloads) to feed into the compiled programs, along with the expected outputs;
3. "Rules" for compiling and running the benchmark programs, and (more notably) for reporting your benchmarking results; and
4. Sample reports, showing by example how results should be reported.

2.1 Our (non-)motivations in creating this suite

Benchmarking is a delicate art and science, and it's hard work, to boot. We have quite limited goals for the `nofib` suite, are hoping for lots of help, and are prepared to overlook considerable shortcomings, especially at the beginning.

2.1.1 Motivations.

- Our main *initial* motivation is to give functional-language implementors (including ourselves) a common set of "real Haskell programs" to attack and study. We encourage implementors to tackle the problems that *actually* make Haskell programs large and slow, thus hastening solutions to those problems.

  And of course, because the benchmark programs are shared, it will be possible to *compare* performance results between systems running on identical hardware (e.g., Chalmers HBC vs. Glasgow Haskell, both running on the same Sun4). Racing is the fun part!

- Our *ultimate* motivation for this benchmark suite is to provide "end users" of Haskell implementations with a useful *predictor* of how those systems will perform on their own programs.

  The initial `nofib` suite will have no value as a predictive tool. Perhaps those with greater expertise will help us correct this. If necessary, we will gladly hand over "the token" for the suite to a more disinterested party.

- We are very keen on (some might say "paranoid about") readily-accessible *reproducible* results. That is the whole point of the "reporting rules" elsewhere in this paper.

  Good-but-irreproducible benchmarking results are very damaging, because they lull implementors into a false sense of security.

- After the initial pre-release of the suite, which will be for (possibly major) debugging, we intend to keep the suite *stable*, so that sensible comparisons can be made over time.

- Having said that, benchmarks must change over time, or they become stale. It is difficult to brim with confidence about the Gabriel benchmarks for LISP systems; they are more than a decade old.

  Being forced to change a benchmark suite can be a mark of success. The SPEC people made substantial changes to their suite in 1992: so much work had gone into compiler tricks that improved SPEC performance results that some tests were no longer useful (notably the `matrix300` test mentioned earlier).

2.1.2 Non-motivations.

We are profoundly uninterested in distilling a "single figure of merit" (e.g., MIPS) to characterise a Haskell implementation's performance.

Initially at least, we are also uninterested in any statistics derived from the raw `nofib` numbers, e.g., various means, standard deviations, etc. You may calculate and report any such numbers—all honest efforts to understand these benchmarks are welcome—but the raw, underlying numbers must be readily available.

An important issue we are *not* addressing with this suite is inter-language comparisons: "How does program X written in Haskell fare against the same program written in language Y?" Such comparisons raise a nest of issues all their own; for example, is it really the "same" program when written in the two compared languages? This disclaimer aside, we do provide the `nofib` program sources in other languages if we happen to have them.

2.2 The Real subset

The `nofib` programs are divided into three subsets: Real, Imaginary, and Spectral (somewhere between Real and Imaginary).

The Real subset of the `nofib` suite is by far the most important. In fact, we insist that anyone who wishes to report any results from running this suite (in whatever form) must first distribute their complete, raw results for the Real subset in a public forum (e.g., available by anonymous FTP).

The programs in the Real subset are listed in Table 1. Each one meets most of the following criteria:

- Written in standard Haskell (version 1.2 or greater).

- Written by someone trying to get a job done, not by someone trying to make a pedagogical or stylistic point.

- Performs some useful task such that someone other than the author might want to execute the program for other than watch-a-demo reasons.

- Neither implausibly small or impossibly large (the Glasgow Haskell compiler, written in Haskell, falls in the latter category).

Program	Description	Origin
`anna`	Strictness analyser	Julian Seward (Manchester)
`calc`	arbitrary-precision calculator	Liang & Mirani (Yale)
`compress`	Text compression	Paul Sanders (BT)
`fluid`	Fluid-dynamics program	Xiaoming Zhang (Swansea)
`gamteb`	Monte Carlo photon transport	Pat Fasel (Los Alamos)
`gg`	Graphs from GRIP statistics	Iain Checkland (York)
`hpg`	Haskell program generator	Nick North (NPL)
`infer`	Hindley-Milner type inference	Phil Wadler (Glasgow)
`lift`	Fully-lazy lambda lifter	David Lester (Manchester) & Simon Peyton Jones (Glasgow)
`maillist`	Mailing-list generator	Paul Hudak (Yale)
`mkhprog`	Haskell program skeletons	Nick North (NPL)
`parser`	Partial Haskell parser	Julian Seward (Manchester)
`pic`	Particle in cell	Pat Fasel (Los Alamos)
`prolog`	"mini-Prolog" interpreter	Mark Jones (Oxford)
`reptile`	Escher tiling program	Sandra Foubister (York)
`veritas`	Theorem-prover	Gareth Howells (Kent)

Table 1: `nofib` benchmarks: Real Subset

- The run time and space for the compiled program must neither be too small (e.g., time less than five secs.) or too large (e.g., such that a research student in a typical academic setting could not run it).

Other desiderata for the Real subset as a whole:

- Written by diverse people, with varying functional-programming skills and styles, at different sites.
- Include programs of varying "ages," from first attempts, to heavily-tuned rewritten-four-times behemoths, to transliterations-from-LML, etc...
- Span across as many different application areas as possible.
- The suite, as a whole, should be able to compile and run to completion overnight, in a typical academic Unix computing environment.

2.3 The Spectral subset

The programs in the Spectral subset of `nofib`—listed in Table 2—are those that don't quite meet the criteria for Real programs, usually the stipulation that someone other than the author might want to run them. Many of these programs fall into Hennessy and Patterson's category of "kernel" benchmarks, being "small, key pieces from real programs" [7, page 45].

2.4 The Imaginary subset

The Imaginary subset of the suite is the usual small toy benchmarks, e.g., `primes`, `kwic`, `queens`, and `tak`. These are distinctly unimportant, and you

Program	Description	Origin
`boyer`	Gabriel suite 'boyer' benchmark	Denis Howe (Imperial)
`cichelli`	Perfect hashing function	Iain Checkland (York)
`clausify`	Propositions to clausal form	Colin Runciman (York)
`fish`	Draws Escher's fish	Satnam Singh (Glasgow)
`knights`	Knight's tour	Jon Hill (QMW)
`life`	Game of Life	John Launchbury (Glasgow)
`mandel`	Mandelbrot sets	Jon Hill (QMW)
`minimax`	tic-tac-toe (0s and Xs)	Iain Checkland (York)
`multiplier`	Binary-multiplier simulator	John O'Donnell (Glasgow)
`pretty`	Pretty-printer	John Hughes (Chalmers)
`primetest`	Primality testing	David Lester (Manchester)
`rewrite`	Rewriting system	Mike Spivey (Oxford)
`scc`	Strongly-connected components	John Launchbury (Glasgow)
`sorting`	Sorting algorithms	Will Partain (Glasgow)

Table 2: `nofib` benchmarks: Spectral Subset

may get a special commendation if you ignore them completely. They can be quite useful as test programs, e.g., to answer the question, "Does the system work at all after Simon's changes?"

3 Rules for running and reporting

Glasgow will provide the `nofib` program sources, as well as input workloads and expected outputs. (We will also provide some "scaffolding" to make it easier to run the benchmarks.)

Anyone can then run the benchmark programs through their Haskell system. The "price" for using the benchmark suite is that you must follow our rules if you report your results in any public forum, including any publication.

In the big-money-on-the-line world of the SPEC suite, the running and reporting rules are complicated and arcane. That's because there are many people who would rather be sneaky than do honest work to improve their system's performance. For now, we assume that functional programmers are more noble creatures; the `nofib` rules are therefore quite simple.

The basic reporting principle is: You must provide enough information and results that someone with a similar hardware/software configuration can *easily* duplicate your results.

The most important specific `nofib` reporting rule is: if you wish to report or publish some results from running some part of the `nofib` suite, then you must first "file" a complete set of how-I-did-it/what-I-got information for the entire Real subset of programs, in some public forum (a newsgroup, mailing list, an anonymous-FTP directory, ...). Thereafter, you may claim whatever you like, the idea being that people can look up your "filed" information and laugh at you if you're making unsubstantiated claims.

We are not insisting on these rules because we like playing lawyer. We intend as little hindrance as possible to creative, honest uses of this suite.

There are more details about the reporting rules in the version of this paper that is distributed with the suite.

4 Concluding remarks

Inattention to benchmarking is not just sloppy, it ends up as self-delusion. Assertions that various functional-languages compilers "... generate code that is competitive with that generated by conventional language compilers ..."[2] are *simply false* by any common-sense measure; what's more, when they are repeated by Respected People, they are downright harmful: they detract from the urgency of building better implementations.

By introducing the `nofib` suite of Haskell programs, we hope for an immediate payoff, simply by giving all Haskell implementors a common set of sources with which to race each other. We also hope that we are setting the foundation for a sound predictor of Haskell-system performance.

This suite follows the general trend away from "plot-the-characteristics benchmarking" and towards "lazy, systems benchmarking," of which the SPEC suite is the most prominent example. This approach to benchmarking gives the greatest credence is given to gross system behaviour on sizable, real programs.

Comments on this paper and on the `nofib` suite itself are most welcome. Contributions of substantial functional programs that could be added to the suite would be even more welcome! I can be reached by electronic mail at `glasgow-haskell-request@dcs.glasgow.ac.uk`.

Haskell-related things, including the `nofib` suite, can be retrieved by anonymous FTP from `ftp.dcs.glasgow.ac.uk`, in `pub/haskell/glasgow`. The sites `nebula.cs.yale.edu` and `animal.cs.chalmers.se` usually have copies as well (in the same directory).

An up-to-date version of this paper will be included in the `nofib` distribution. There is also a top-level `README`, which is the first file you should consult.

Acknowledgements. My thanks to John Mashey for his many fine articles in `comp.arch` that promote sensible benchmarking, and to Jeff Reilly for providing information about SPEC. Vincent Delacour, Denis Howe, John O'Donnell, Paul Sanders, and Julian Seward were among those who provided helpful comment on earlier versions of this paper. Of course, we are *most* indebted to those people who have let their code be included in the suite.

References

[1] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *Computer Journal*, 32(2):127–141, April 1989.

[2] Adrienne Bloss, P. Hudak, and J. Young. An optimising compiler for a modern functional language. *Computer Journal*, 32(2):152–161, April 1989.

[2] Citation withheld to protect the guilty!

[3] Thomas M. Conte and Wen-mei W. Hwu. A brief survey of benchmark usage in the architecture community. *Computer Architecture News*, 19(4):37–44, June 1991.

[4] Richard P. Gabriel and Larry M. Masinter. Performance of Lisp systems. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 123–142, Pittsburgh, PA, August 15–18 1982.

[5] Brent Hailpern, Tien Huynh, and Gyorgy Révész. Comparing two functional programming systems. *IEEE Transactions on Software Engineering*, 15(5):532–542, May 1989.

[6] Pieter H. Hartel. Performance of lazy combinator graph reduction. *Software—Practice and Experience*, 21(3):299–329, March 1991.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[8] Lynn Pointer, editor. Perfect report 2. CSRD Report 964, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, March 1990.

[9] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[10] S. C. Wray and J. Fairbairn. Non-strict languages—programming and implementation. *Computer Journal*, 32(2):142–151, April 1989.