Grouping Common Lisp Benchmarks

Bignums and Consing and Vectors, Oh My!

Christophe Rhodes*

Centre for Computational Creativity
City University
Northampton Square
London EC1V 0HB

April 5, 2004

Abstract

Compiler benchmarks provide a means for compiler writers to measure their performance, as well as for potential users to estimate whether a given implementation will suit their performance needs. We present a simple application of clustering algorithms in the programmatic examination of a freely-available suite of benchmarks to estimate the similarity of the codepaths tested, and find a grouping of benchmarks with respect to the implementation.

1 Introduction

Measuring the performance of Lisp systems is an important art, particularly given the common yet woefully ill-informed perception that Lisp is a 'slow language'. It is therefore not surprising to find that there are many studies, from the relatively old [1] to more recent presentations [2] of comparative performances on suites of benchmarks.

The *de facto* standard performance benchmark suite for Common Lisp today is the `cl-bench` [3] suite, which aggregates among other benchmarks the well-known Gabriel benchmarks, bignum-exercising code due to Bruno Haible, hash table stressing, and tests of various aspects of CLOS. The benchmarks within the suite are classified by broad category for presentation as a series of graphs, but this classification is performed by hand, with a view to the history of the benchmarks and to their outward form, rather than to any underlying behaviour of the benchmarks. We argue here for a more automatic classification based on past performance characteristics, so that users and implementors alike can determine whether additional benchmarks give any new information, and whether a given benchmark measures what it claims to measure.

*`crhodes@city.ac.uk`

Classification of benchmarks in this fashion is likely to be useful both to users and to implementors. The gain for users is indirect: while a classification in terms of the outward appearance of the code, or by history, is valid from the user's point of view, it is also of importance to know which benchmarks test independent subsystems, so that the perception of implementations' performance is not biassed by a preponderance of benchmarks within a suite testing any particular codepath over the others.

The benefit to implementors is more immediate; much as it is nowadays considered *de rigeur* in any sizeable software project to have a regression test suite (to manage the complexity of interacting systems), it is useful to have a suite testing execution time, so that avoidable degradations in performance can be caught early and remedied. The classification, or rather the knowledge of which benchmarks exercise which codepaths in a compiler, assists the implementor in pinpointing the cause of any performance degradation that may occur.

2 Analysis

We model each timing result t_{ij} for a benchmark i and implementation revision j as the weighted sum of conceptual components (or codepath costs) due to the implementation f_{kj}, so that

$$t_{ij} = a_{ik} f_{kj} + \text{noise}. \tag{1}$$

For a given implementation, then, a benchmark is characterized by the a_{ik} coefficients, and the change in timings over versions is due to the implementation improving (or otherwise) its performance in the components f_{kj}. This statement is only valid on the assumption that the implementation does not change its evaluation strategy in any pervasive way; for instance, the introduction of an interpreter to a compiler-only implementation might be modelled as addition of a new f_{kj}; equivalently, however, this 'new' f_{kj} can be thought of as having been there all along, but until the interpreter's addition having performance characteristics degenerate with the old evaluation strategy.

A measure of the closeness of benchmarks, then, is the similarities of their response; if we assume that changes between revisions j are sufficiently atomic as to have only one effect on the components f, then the change in t is a direct measure of the weights a, and therefore the benchmarks' underlying similarity. Since revisions are not necessarily atomic with respect to the components, and since statistical noise contaminates timing values, we need to disentangle sources.

One way of doing so is to estimate clusters within the data. Here we present the application of hierarchical clustering as a simple method for data visualization and estimation. Given timing results $t_{ij} \pm \sigma_{ij}$, where σ_{ij} is our estimate for the standard error on t_{ij}, we construct the delta vectors δ_{ij} by

$$\delta_{ij} = \begin{cases} \frac{t_{ij+1} - tij}{\max_j t_{ij}} & t_{ij+1} - t_{ij} > n\sqrt{\sigma^2_{ij+1} + \sigma^2_{ij}} \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

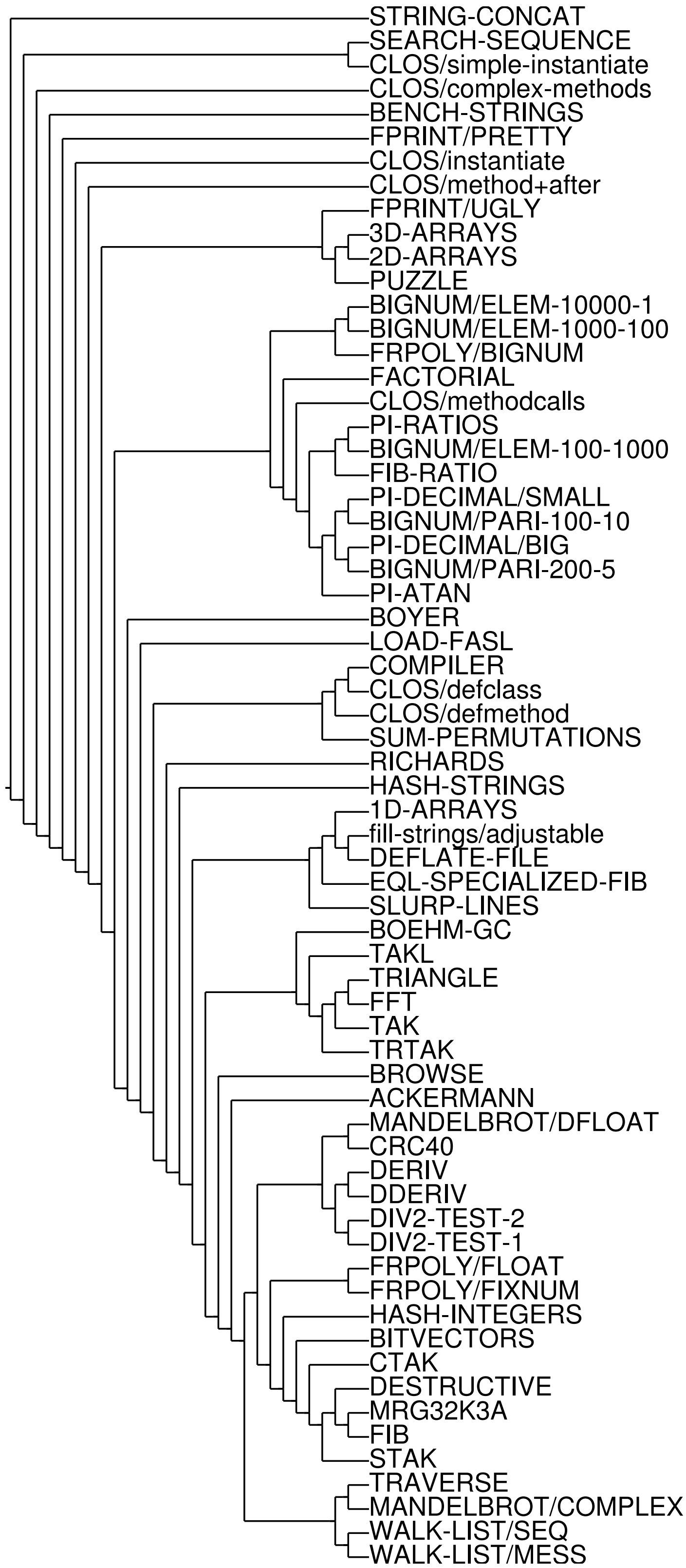

Figure 1: Benchmark clustering, based on results with SBCL versions

Arrays	FPRINT/UGLY – PUZZLE
Boxed Numbers	BIGNUM/ELEM-10000-1 – PI_ATAN
Compiler	COMPILER – SUM-PERMUTATIONS
Vectors	1D-ARRAYS – SLURP-LINES
Consing	MANDELBROT/DFLOAT – DIV2-TEST-1
Unprobed	TRAVERSE – WALK-LIST/MESS

Table 1: Tentative benchmark cluster allocations and characterizations

where N is an adjustable parameter measuring the confidence required for a step to be treated as significant. We can then cluster the benchmark δ_{ij} against each other using a Euclidean distance metric to give a visualization of benchmark relationships.

We performed this clustering on benchmark results, obtained on a 1GHz Pentium III with 1Gb RAM running FreeBSD, from versions of SBCL [4] dating from May 2001 to March 2004, using data from all released versions in this period (approximately one every month) as well as several series of revisions between releases[1]; n was set to 3, echoing scientific convention. Note that the monthly data are probably not from sufficiently atomic changes for our assumption, above, to hold. The results are shown[2] in figure 1.

From the figure, we can make some tentative identifications. In typical clustering applications, to get k clusters we simply cut the k longest links in the dendrogram. However, this is not so appropriate to our aims here; we want to identify tightly-spaced regions of multiple benchmarks.

Table 1 shows some tentative allocations of benchmarks to clusters of strong similarity, as well as speculative characterizations of these clusters' dominant factor for SBCL; while some of the elements of clusters are probably erroneous – it is difficult to imagine CLOS/methodcalls being a benchmark testing boxed numbers intensively, for instance – some of the other groupings are revealing: detecting that SBCL's performance in CLOS/defclass and CLOS/defmethod is related to its performance in the COMPILER benchmark.

Of course, one weakness of this method is that it cannot assign to any cluster a benchmark which has not exhibited any significant changes in times over the history; in this investigation, for SBCL, the TRAVERSE, MANDELBROT/COMPLEX, WALK-LIST/SEQ and WALK-LIST/MESS belong in this "Unprobed" category. Another weakness is the loss of distance information between clusters to simply a binary 'nearest' relation. We can nevertheless confirm that at least some of the clusters we have identified are robust, by comparing with a simple alternative visualization: a graph representing clusters identified solely by distance (figure 2).

[1] see `http://sbcl.boinkor.net/benchmark/` for the raw data.

[2] Graphical representation courtesy of McCLIM's implementation of the CLIM Graph Formatting protocol.

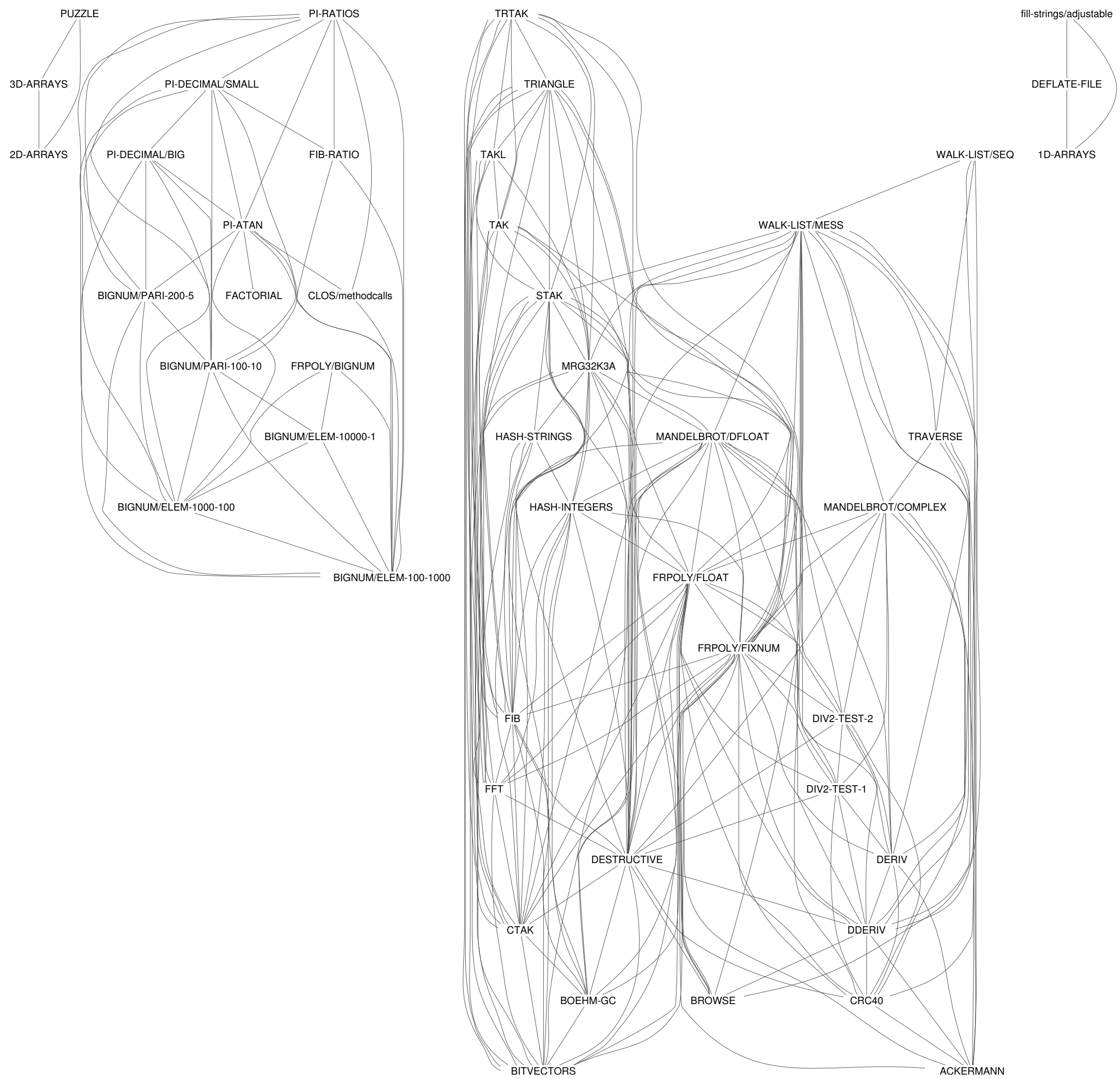

Figure 2: Simple distance-based clustering of benchmarks.

3 Further Work and Conclusions

It is clear that this analysis can yield valuable information to people wishing to interpret results of benchmarks; while this clustering can in no way detract from the raw measured performance, it can help to explain particular characteristics in an implementation, and to understand differences between implementations on the same hardware. It is also clear that the visualisation of clusters from this scheme is imperfect and largely dependent on a fair amount of familiarity with the implementation in question; a higher resolution on the revision scale would help with this, but it is probably necessary to develop a complementary visualisation technique to identify essentially degenerate benchmarks.

One continuation of potential interest would be to repeat this analysis using benchmark results from a different implementation. This presents an interesting technical challenge, however, in that for practicality the implementation has to be available with incremental changes at a resolution of no greater than a month, and ideally on a finer scale. For practical reasons, this therefore reduces the other potential ANSI Common Lisp implementations for this investigation to those which can be reliably built from source with no external thought[3]: in essense, to CLISP. This would nonetheless be an interesting comparison to make, as the two implementations share no common history.

In this investigation, we benefitted from Lisp's dynamicity while investigating suitable measures and distance metrics, and also from CLIM's extensible graphical tools (McCLIM's Postscript extensions for preparing figure 1 and the McCLIM Listener application for rapid feedback). The benchmark results themselves, as we have observed, should be taken with a pinch of salt, not only because they do not necessarily measure what an application programmer will use, but also because different benchmarks may be measuring similar underlying processes.

Acknowledgments

I am grateful to Andeas Fuchs for writing programs to execute the benchmark suites and for performing the runs, and to Eric Marsden for useful discussions, and for assistance in preparing figure 2.

References

[1] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

[2] Rainer Joswig. Lisp and Mac OS X. In *International Lisp Conference Proceedings*, 2003.

[3] Eric Marsden. Common Lisp Benchmarking Suite. Available at `http://www.chez.com/emarsden/downloads/cl-bench.tar.gz`.

[4] Christophe S. Rhodes et al. Steel Bank Common Lisp User Manual. in preparation.

[3] This quality is part of SBCL's *raison d'être*.