Extended Abstract — Unibench: The Swiss Army Knife for Collaborative, Automated Benchmarking

Daniel Rolls, Stephan Herhut, Carl Joslin, and Sven-Bodo Scholz

School of Computer Science, University of Hertfordshire, AL10 9AB, UK
{D.S.Rolls,S.A.Herhut,C.A.Joslin,S.Scholz}@herts.ac.uk

Benchmarking is a significant part of programming language and architecture research. To show the merit of a new idea — a new programming language or merely a new compiler optimisation — an evaluation of its impact on runtime behaviour is essential. Yet, the benchmarking process itself has very little research value. Instead, it is a tedious and labour-intensive task.

The challenges start when choosing an appropriate set of programs to be analysed and when designing an adequate setup for the experiment. The former requires, apart from the selection process itself, a search for existing implementations of the selected programs or even an implementation from scratch. For the latter, the setup of the environment, the desired program parameters, tool-chain settings, hardware configurations, etc. need to be chosen. Once all these decisions have been made, the actual experiments need to be performed.

This usually is where the tedious work begins. Systematic experiments require a range of single experiments to be performed in a controlled environment. Such experiments bind precious resources. Firstly, the controlled environment commonly requires exclusive access to some hardware resource. This hardware often is purchased for the sole purpose of benchmarking to minimize the impact of benchmarking on other day-to-day work. Secondly, a researcher's attention is required to produce the binaries, perform the experiments and collect the data. However, to rule out human error and thus increase consistency, and to liberate oneself from the dull, repetitive task of experimentation, the measurement process is often automated to a certain degree.

The attempt to at least semi-automate the experimentation process, in the authors' experience, is often a painful one. Errors that occur during a single experiment may inhibit further experiments to be performed. We may see failures in tool-chains, inappropriate problem sizes that exhaust the available resources, errors in the invocation scripts, defects in the hardware or software, etc. Such errors in most cases imply changes in the benchmarking scripts which, in turn, often requires *all* experiments to be repeated in order to guarantee consistency of the investigation. This process can repeat itself resulting in a significant prolongation of the overall benchmarking process.

Once all measurements have been completed, the actual task of interpreting the results can begin. However, trying to find explanations for peculiar artefacts in the measurements often results in the need for further experiments such as repeating the experiments with slightly modified setups or on different hardware-configurations. This leads to an iteration of the measurement process where scripts need to be adjusted and experiments need to be rerun.

After several iterations of the above cycle, the final set of experiments needs to be documented in a way that enables other researchers to repeat the experiments and, hopefully, to obtain the same results. In our experience, this is close to impossible. Source code must be archived and made available publicly. Archiving over long periods is difficult since the data may often be in one of many organisations and over time organisations evolve and disband. Centralisation is desired here.

Even with good documentation and public source code and where the particular version of tool-chain used is still available, the executing machinery is almost never available in the same form as it was for the original experiment. The same applies for the painfully generated scripts for automating the measurements. Just within a single research group it can be difficult to pass on these things because such benchmarking scripts tend not to be flexible enough to cover a wider range of application scenarios and thus are further adapted over time. Benchmarking in a cross-institution, cooperative fashion takes these difficulties one level further. Each institution may have its own set of benchmarking suites, hardware resources and scripts for automating experiments. Making these available across institutions and keeping the different versions consistent is a difficult, if not impossible task.

We believe the time to overcome this ad-hoc approach of benchmarking is overdue. Hence we have developed a tool that helps researchers to archive, document, and perform benchmarking experiments in a much more structured, efficient and collaborative way. This extended abstract presents the universal benchmarking tool Unibench. It is a tool for benchmarking compilers, architectures and algorithms and for dissemination of results that enables efficient and accurate experimentation with pooling of resources. Unibench can coordinate and automate the running of experiments, archive measurement methodologies and their results and disseminate this information to the public. The tool tracks configurations of compilers and host machines and can choose implementations of benchmarks with different inputs to run on these configurations. Custom measurement scripts allow high degrees of flexibility.

In Unibench benchmarks are abstract specifications for implementations that specify the inputs they should expect, the outputs they should produce from these inputs and how the implementations are allowed to act. Benchmarks, benchmark implementations and runtime inputs for benchmarks persist and Unibench even stores the code used to perform a measurement itself. We consider this an important feature.

Compilers and standard libraries used for experiments are recorded but reside on the systems upon which the compilers are run rather than on Unibench itself. Compilers are rarely self-contained, they need to be installed and can be configured in different ways and so belong as part of a setup on an operating system. We consider the ability to run experiments on architectures and with compilers and libraries that we have no control over both convenient, since measurements from these systems will often be interesting, and a pragmatic requirement.

When benchmarking, changing compiler flags is a very common process so compiler flags available for compilers persist along with a record of which flags were used for each experiment.

Finally machines upon which experiments are run have to be registered with Unibench and the specifications of the machines are documented on Unibench.

Any information stored in the database upon which a result depends must be kept for archiving purposes. Users however invariably wish to make improvements to anything they save. In Unibench the general philosophy is to write once. When mistakes are made it is better to add something new than to change something. This philosophy helps to remove concerns over validity of results. Names can still be corrected and descriptions can be improved but uploaded files and version numbers should in general stay in the database. Users can delete their own uploads immediately after a mistake but once experiments have been recorded we try to keep results.

To avoid removing useful results in the future any saved, static, interpreted representations of results from Unibench depend upon these results and the results cannot be removed, even by an administrator, without first removing the interpreted representations they depend upon. It is also possible to *pin* results in Unibench by declaring that a set of results was quoted somewhere and needs to stay archived (along with its dependencies) so that anybody interested in claims made in the publication can see the experiment for themselves.

Core to unibench are measurements themselves. Reusable measurements may include wall clock time, processor clock cycles and processor instruction counts. If we wish to make Unibench universal we cannot impose the metrics that should be measured or the means by which these measurements should take place. These scripts are used to perform measurements without any user intervention. This helps to avoid mistakes or accusations of mistakes in measuring. This is important for arguing about impartiality during the measurement process.

Unibench allows scripts to be uploaded to perform arbitrary measurements. These scripts typically return a single numerical result. Sometimes however this is inconvenient since more information can be collected from a single run. For this reason arrays of floating point numbers of any dimensionality are supported. For complete flexibility the scripts are passed the compiled program to run along with any runtime input. This allows the scripts to run the programs as many times as they wish. Scripts are copied to target machines when they are needed.

For compiler writers we model versioning of compilers to allow comparisons between different versions of compilers. This allows comparisons of results from different compiler versions and regression testing.

The modelling of restrictions on versions is supported like source code only working with older or newer versions of a standard library or compiler.

A de facto standard interface already exists for compiler calls e.g. ‘gcc mysource.c -O3.’ We provide a compatible interface so that compilers that support the source code passed as the first argument and flags passed as the latter arguments can easily be accessed by Unibench without any need to write a script to intercept and rearrange arguments. Most C and Fortran compilers can be exposed to Unibench as a symbolic link to the compiler binaries. The names of compiler scripts are uniquely determined by compilers and their versions. Unibench just needs to be told which compiler versions and standard libraries exist on which machines and it will look in a predefined place for an appropriate compiler script.

Our decision to proactively perform experiments has meant that scheduling of jobs has to be taken into consideration. We try to give higher priority to newer entries in the database since they are more likely to be considered interesting and because it means that when somebody uploads a new implementation for benchmarking experiments can be run on it promptly.

We cannot assume however that any heuristic we use will correctly anticipate how important each implementation of a benchmark is considered by the owner of a machine. For this reason, machine owners have the optional ability to specify that something should not run or that it is of low, medium or high priority. This can be specified for whole benchmarks as well as for groups of benchmarks. Priorities can also be overridden so, for example, a benchmark could be set to low priority and an individual implementation of that benchmark could be set to high priority, overriding the default for that benchmark. The scheduled runs for each machine can be browsed whether or not that machine is enabled. The order of experiments run on each machine is determined primarily by the prioritisation mechanism and subsequently by the age of the entries in the database determining the result.

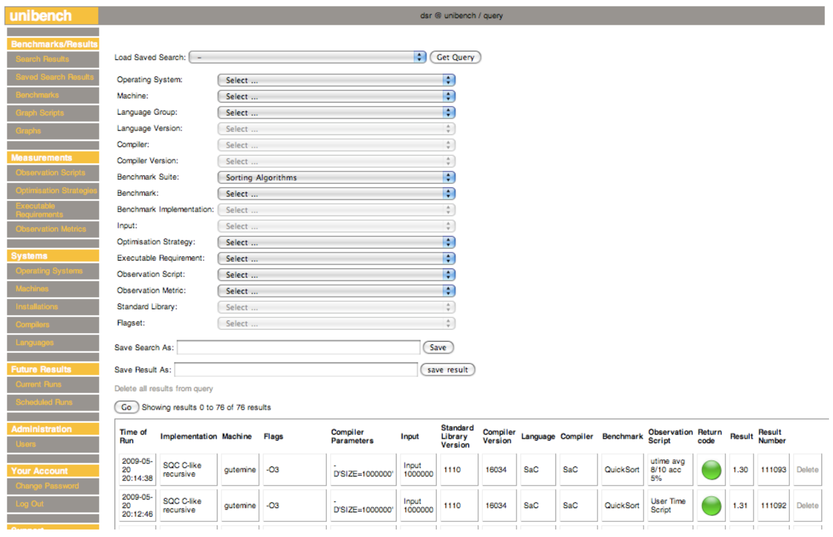
Allowing arbitrary code and arbitrary measurement systems to run on remote systems has obvious security implications. If these machines have means of establishing external connections or accessing or modifying any data that should be protected then the benchmarks and observation scripts could be harmful. For auditing purposes users have accounts and ownership of uploaded content is tracked. Every user has an assigned privilege level determining their level of access. Disregarding administrators, the more highly privileged users can influence what is run and where, whereas less privileged users can just upload benchmarks and benchmark implementations.

Unfortunately, current publicly exposed databases of arbitrary benchmarking results tend to only be of use to those who already know what they are looking for. Some, like Netlib's Perfomance Database Software [1], have links to papers documenting standards but these links to separate pdf documents are the only guidance for the user available. This can only discourage the casual browser.

In Unibench benchmarks are placed into organised categories that we call *benchmark suites*. These can be nested to provide a hierarchy organising the benchmarks. We provide means for users to document benchmark implementations, benchmarks and benchmark suites. The former allows discussion of the details of the specific implementation and why it is interesting to benchmark. Benchmarks should be seen as a contract defining what is and is not allowed from an implementation and what inputs are expected and outputs should be produced. Benchmark suites are broader and allow the areas for which benchmark contracts are written to be introduced.

All this information allows us to create browsable pages of user-customized text to make the system more like a website and to allow it to be used for disseminating information. This could be considered just as effective a means of reaching a wider audience as a publication — all of these pages are indexed by Google.

Special-purpose benchmarking tools like The Computer Language Benchmarks Game [2] can be designed with careful consideration of the kinds of results that users will be interested in viewing. With Unibench this is not so easy: Unibench's design as a universal tool needs support for queries on an arbitrary selection of fields. However, a slow or difficult to use result retrieval system would severely hamper the ease of result dissemination with Unibench.



Load Saved Search: -

Operating System:	Select ...
Machine:	Select ...
Language Group:	Select ...
Language Version:	Select ...
Compiler:	Select ...
Compiler Version:	Select ...
Benchmark Suite:	Sorting Algorithms
Benchmark:	Select ...
Benchmark Implementation:	Select ...
Input:	Select ...
Optimisation Strategy:	Select ...
Executable Requirement:	Select ...
Observation Script:	Select ...
Observation Metric:	Select ...
Standard Library:	Select ...
Flagset:	Select ...

Save Search As:

Save Result As:

Showing results 0 to 76 of 76 results

Time of Run	Implementation	Machine	Flags	Compiler Parameters	Input	Standard Library Version	Compiler Version	Language	Compiler	Benchmark	Observation Script	Return code	Result	Result Number	
2009-05-20 20:14:38	SQC C-like recursive	gutemine	-O3	-D'SIZE=1000000'	Input 1000000	1110	16034	SaC	SaC	QuickSort	utime avg 8/10 acc 5%	●	1.30	111093	Delete
2009-05-20 20:12:46	SQC C-like recursive	gutemine	-O3	-D'SIZE=1000000'	Input 1000000	1110	16034	SaC	SaC	QuickSort	User Time Script	●	1.31	111092	Delete

Fig. 1. A screenshot of the result browsing facilities in Unibench.

These problems were solved by opening up many fields to the user for narrowing down results; a dependency mechanism between fields was built to allow instantaneous filtering of available selections from fields. With this mechanism we've built a system for retrieving results where users are never forced to wade though lists of hundreds of elements or to manually enter arbitrary text into text boxes. Figure 1 shows the result browsing facilities in Unibench. A query for results can be narrowed down by using the dropdown boxes and hitting 'go' causes the results to be displayed underneath. The circle in the results table is coloured green to indicate a successful run and red to indicate an error result. Errors like compilation failures, programs exiting with errors and custom error checking in scripts are reported to the user as tooltips on these circles. Users are not troubled with recoverable errors like network failures.

Whilst showing code and raw results is extremely important for openness we believe that for public dissemination it is particularly important that interpreted results are made available in the form of graphs. In various technology companies the authors have seen custom made systems producing graphs for internal use only.

For Unibench any graphing solution must be universal. We decided that the only way to allow for universal control over a dataset is to allow users to write code to manipulate the dataset as they see fit. We would consider an array processing language appropriate for this and chose the SaC [3] functional, array processing language which was originally created by one of the authors and with which all of the authors are involved.

Any dynamic manipulation of datasets must take account of the fact that the data might change. The data must be manipulated in a generic way that is reusable as the data changes and for future datasets returned from queries.

Users can specify a query from which to produce a graph (or other output). They are then provided with information on the dataset returned by the query and can request an n-dimensional datacube in SaC to manipulate. Carefully written code can be reused for the same query and even for different queries after more results have been produced. We hope this encourages collaboration and means users will over time produce appropriate graphs for their own requirements rather than compromising by using whatever limited graphs format we give them. Currently graphs are produced by outputting scripts for Gnuplot [4] but we have purposely designed the system in such a way that we can add support for other graph generators without too much work.

Bringing these principles together within Unibench has led to a tool which we consider the swiss army knife for benchmarking. All experiments are initiated by Unibench itself. This enforces complete and formal specifications of *all* parameters required to run an experiment. As a nice side-effect, this requirement facilitates collaborative experiments, where specifications come together from various parties. The use of the Internet as a communication layer renders geographic locality irrelevant. Furthermore, Unibench enables direct reuse of results, i.e., experiments can be easily extended by third parties. Apart from the sharing of experiments, Unibench also enables the sharing of hardware resources to perform experiments on. This is of particular interest when experiments are to be run on novel platforms which are not generally available yet.

So far, we have used the tool mainly in the context of the EU FP7 project Apple-CORE [5] that funded its main development. However, it very quickly developed a dynamics of its own. After a few months, we have eight machines from three institutions in the Netherlands, the UK, and Canada contributing to Unibench. Thousands of experiments have been run, ranging from instruction counts and cache miss rates, over memory and runtime uses, to FLOPS performed. As diverse as the measurements have been are the tool chains that have been investigated. These range from everyday compilations / executions to emulator runs that interpret the result of several different compilation stages.

From these experiences, we can see that Unibench has a far wider applicability than anticipated. An example for such a future extended use are regression tests. The modelling capabilities for experiments in Unibench, which are based on the idea of execution requirements, enable users to easily integrate result checking mechanisms of various kinds. The only extension of the system that is required to achieve this is a facility for keeping benchmark results in the database that underlies Unibench.

Unibench is publicly accessible at

`http://unibench.apple-core.info`

A guest to the website without a user account can browse through suites of benchmarks and view their specifications, implementations, accepted inputs and results from various experiments.

Acknowledgments A first prototype of Unibench was developed by Maxime Castelein and Julien Viera from the Polytech'Lille Engineering School, France, during a summer internship at the University of Hertfordshire.

The reimplementation that led to the current system was funded by the European Union Framework 7 project Apple-CORE grant no. FP7-215216.

References

1. LaRose, B.H.: The development and implementation of a performance database server. Master's thesis, University of Tennessee, Knoxville, TN, USA (1993)
2. Debian.org: The computer language benchmarks game (2009) `http://shootout.alioth.debian.org` (last accessed 2009-07-02).
3. Scholz, S.B.: Single assignment c — efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13** (2003) 1005–1059
4. Gnuplot: Gnuplot website (2009) `http://www.gnuplot.info` (last accessed 2009-07-10).
5. Apple-CORE: Apple-core website (2009) `http://apple-core.info` (last accessed 2009-07-02).