Unibench: A Tool for Automated and Collaborative Benchmarking

Daniel Rolls, Carl Joslin and Sven-Bodo Scholz
University of Herfordshire, Hatfield, UK
{*d.s.rolls,c.a.joslin,s.scholz*}*@herts.ac.uk*

***Abstract*—We have identified the need for a universal benchmarking tool that enforces consistency as well as proper documentation. Enforcing these aspects without restricting the tool's applicability poses a major challenge.**

This paper introduces a tool for coordinating the running of experiments on remote machines. A simple web interface allows for source code to be submitted. Experiments are run and results are publicly disseminated via a web interface without user intervention. The system has already enabled sharing of resources internationally and good scientific inquiry.

I. Introduction

Those with experience in performing any kind of dynamic code analysis or runtime measurements know that this is more tedious than it appears. The challenges are choosing an appropriate set of programs to be analysed and choosing an adequate setup for the experiments. A setup includes such choices as program parameters, tool-chain settings and hardware configurations.

Systematic experiments require a range of measurements to be taken, each of which utilises a hardware resource exclusively. Typically, scripts are used to trigger these experiments automatically. Repeated manual invocation increases the chance of experimental errors, and requires the researcher's constant attention.

In the authors' experience, even semi-automation can be troublesome. We may see, for example, failures in tool-chains, problems which require excessive resources, errors in scripts or defects in the hardware or software. In most cases, such errors imply changes in the benchmarking scripts which, in turn, often requires *all* experiments to be repeated to guarantee the consistency of the investigation.

Errors often result in the need for further experiments such as repeating runs with slightly modified setups. This leads to an iteration of the measurement process where scripts need to be adjusted and experiments repeated.

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 equivalent results. In our experience, this is near impossible. Even with good documentation, 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 laboriously generated scripts for automating the measurements. Even within a single research group it can be difficult to share these scripts which tend not to be flexible enough. Cross-institution cooperative benchmarking compounds these difficulties.

II. Overview Of Unibench

We have developed Unibench to help researchers to document, perform experiments and archive in a more structured, efficient and collaborative way than earlier tools allowed. Users can submit scripts to perform *any* experiment, which may perform any kind of dynamic runtime analysis for which command line tools exist.

Unibench tackles the problems of human error when running experiments, the lack of sharing of resources and common infrastructures for benchmarking, and the unrepeatablility of many published experiments; without long term archiving unrepeatability becomes inevitable even in the most diligent research group.

Unibench manages source code, compilers and measurement scripts and initiates all possible compilations of source code. It runs all measurement scripts that can be run on the compiled binaries and hence the burden of scheduling jobs is removed from users. The scheduler orders jobs to avoid unnecessary compilation and to give preference to experiments using newer content in the database. A priority mechanism exists to allow users to affect the scheduler when necessary in a simple way.

Unibench automates the running of experiments. Therefore any tool for case studies must have the capability to replay any user interaction simulated from runtime input. These custom inputs during runtime analysis are supported by simply uploading input files through the web interface. It is possible also to upload multiple versions of source files and to observe changes in software over time.

The compilers that Unibench can use may be on machines anywhere in the world. Machine owners can provide Unibench with an ordinary SSH login. The priority mechanism specifies priorities per machine and a rights system restricts which accounts can be used to change priorities, enabling machine owners to control what can be run.

Source code, measurement scripts and different configuration options for compilers are stored centrally and distributed to machines on demand. This encourages sharing, aids reuse and provides transparency. Figure 1 illustrates this remote management of jobs by Unibench. Both static and dynamic code analysis tools [2] for program comprehension

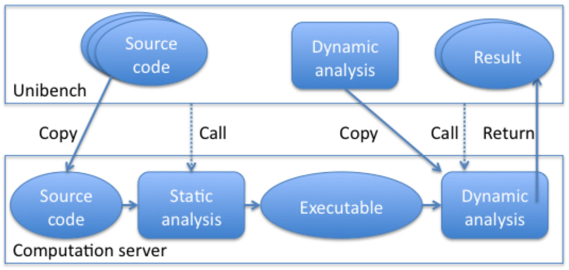

Figure 1. Process flow for Unibench experiments. Compilation and measurement are shown as static and dynamic analysis to emphasise that any kind of static or dynamic analysis is possible.

e.g. software Reconnaissance tools [1] could be used. Any command-line driven tool that produces a numerical metric or indexed array of metrics is viable.

We regret that machine configurations will always change but as new machines and compilers appear, Unibench will automatically run old experiments on the new configurations for comparison with new experiments.

The results Unibench has collected are publicly viewable. An online search form allows users to quickly narrow down search criteria to find specific results. Source code is highly categorised to ensure that this is always manageable.

For interpretation of results a powerful scripting solution exists [3]. Scripts are applied to search queries and are used for common tasks like graph generation. Graphs created via Unibench, like almost everything else in Unibench, are publicly viewable along with their dependencies.

Crucially, all data used in graphs stays in the database, along with source code and measurement scripts that were used when experiments were run. This allows good scientific enquiry since there is no need to email publishers of graphs asking for scripts and source code used for experiments; with a healthy scepticism we can check experiments ourselves.

Unibench uniquely integrates submission of code, automated running of experiments and public dissemination of the results. In this way results are shared and resources pooled in a way that encourages better scientific inquiry. Unibench schedules the running of these experiments but what these experiments are and how they run is left open. This allows research groups with very different methodologies to benefit from the system and even reuse existing scripts and tools for measurements and graphs.

So far the main hurdle to adoption has been encouraging initial setup of custom compilation and measurement phases for users' unique requirements: code submission through the web form has raised few problems.

As a use case, if Unibench were used for profiling, measurement scripts would need to initiate a profiler and could produce a one-dimensional list of method call frequencies indexed by name. Functionality already exists to produce bar graphs for multiple data series where array indexes are on the x axis. Hence this functionality could be used to plot the frequency of method calls side-by-side for different inputs, tools, tool versions or source implementations. Equally, different algorithms for dynamic feature analysis on matrices of results [1], can be implemented on Unibench with scripts. The work from the above cited paper can be wrapped up into a measurement script used on Unibench which calls the tool and returns the result array. Similarly, any automatable tool that given an executable produces a numerical metric is a valid measurement script that can be uploaded.

III. Summary

Once a machine is registered with Unibench the steps to initiate experiments are to install a script to compile and/or statically analyse code and register it with the web interface; to upload to the web interface a measurement script for your desired dynamic analysis; to upload source code via a web form; and to browse and interpret results with scripts on the web site.

Currently, users can upload source code implementations via a simple web interface and, without further effort, various experiments to compare these implementations will be run on systems around the world. Results from these experiments are immediately publicly disseminated, although code with copyright restrictions can be hidden.

Guests to the Unibench website can browse through suites of benchmarks and view their specifications, implementations, accepted inputs and results from various experiments. We want to encourage collaboration amongst computer science research groups in this area, so that the community can share benchmarking data and learn from each other.

Unibench is publicly accessible at unibench.apple-core.info. The system is actively used and runs experiments on machines in the UK, Netherlands, Greece and Canada.

Acknowledgment

Funded by the European FP-7 Integrated Project Apple-core (FP7-215216 — Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs).

References

[1] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 602, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Thomas Gschwind, Johann Oberleitner, and Martin Pinzger. Using run-time data for program comprehension. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 245, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.