Automatic Benchmark Profiling Through Advanced Trace Analysis

Alexis Martin[1,2,3(✉)] and Vania Marangozova-Martin[1,2,3(✉)]

[1] CNRS, LIG, 38000 Grenoble, France
[2] University Grenoble Alpes, LIG, 38000 Grenoble, France
[3] Inria, Grenoble, France
alexis.martin@inria.fr, vania.marangozova-martin@imag.fr

Abstract. Benchmarking has proven to be crucial for the investigation of system behavior and performances. However, the choice of relevant benchmarks still remains a challenge. To help the process of comparing and choosing among benchmarks, we propose a solution for automatic benchmark profiling. It computes unified profiles reflecting benchmarks' duration, function repartition, stability, CPU efficiency, parallelization and memory usage. It identifies the needed system information for profile computation, collects it from execution traces and produces profiles through automatic and reproducible trace analysis. The paper presents the design, the implementation and the evaluation of the approach.

1 Introduction

System performance is a major preoccupation during system design and implementation. Even if some performance aspects may be guaranteed by design using formal methods [1], all systems undergo a testing phase during which their execution is evaluated. The evaluation typically consists in quantifying performance metrics or in checking behavior correction in a set of use cases. In many cases, system performance evaluation does not only consider absolute measures for performance metrics but is completed by *benchmarks*. The point is to use well-known and accepted test programs to compare the target system against competitor solutions.

Constructing a benchmark is a difficult task [2] as it needs to capture relevant system behaviors, under realistic workloads and provide interesting performance metrics. This is why benchmarks evolve with the maturation of a given application domain and new benchmarks appear as new system features need to be put forward. Developers frequently find themselves confronted with the challenge of choosing *the right* benchmark among the numerous available. To do so, they need to understand under which conditions the benchmark is applicable, what system characteristics it tests, how its different parameters should be configured and how to interpret the results. In most cases, the choice naturally goes to the

This work is funded by the SoC-Trace FUI project http://soc-trace.minalogic.net.

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 63–74, 2016.
DOI: 10.1007/978-3-319-43659-3_5

most popular benchmarks. Unfortunately, these are not suitable for all use cases and an incorrect usage may lead to irrelevant results.

In this paper, we present our solution for automatic profiling of benchmarks. The profiles characterize the runtime behavior of benchmarks using well defined metrics and thus help benchmark comparison and developers' choices. The profile computation uses information contained in execution traces and is structured as a deterministic trace analysis workflow. The contributions of the paper can be summarized as follows:

- *definition of unified profiles for benchmarks.* We define profiles in terms of execution duration, function repartition, stability, CPU efficiency, parallelization and memory usage. These are standard metrics and can be easily understood and interpreted by developers.
- *definition of the tools needed to compute the profiles.* We structure the computation as a reproducible workflow with parallel and streaming features. The final workflow is automatic and may be easily applied to different benchmarks or to different configurations of the same benchmark.
- *definition of the data needed for profile computation.* We use system tracing and extract useful data in application-agnostic manner. The application source code is not needed.
- *profiling of the Phoronix benchmarks.* We use our solution to profile the Phoronix Test Suite [3]. The results are obtained on different embedded and desktop platforms.

The paper is organized as follows. Section 2 presents the design ideas behind our solution. Section 3 discusses our implementation by considering the Phoronix use case. Section 4 distinguishes our proposal from related work. Finally, Sect. 5 presents the conclusions and the perspectives of this work.

2 Automatic Profiling of Benchmarks

This section presents the benchmark profiles (Sect. 2.1), the needed data for their computation (Sect. 2.2) and the computation process itself (Sect. 2.3).

2.1 Benchmark Profiles Definition

The profile considered for a benchmark is independent of its semantics and is composed of the following features:

- *Duration.* This metric gives the time needed to run the benchmark. It allows developers to estimate the time-cost of a benchmarking process and to choose between short and long-running benchmarks.
- *CPU Occupation.* This metric characterizes the way a benchmark runs on the target system's available processors. It gives information about the CPU usage, as well as about the benchmark's parallelization.

- *Kernel vs User Time.* This metric gives the time distribution between the benchmark-specific (user) and kernel operations. It gives initial information on the parts of the system that are stressed by the benchmark.
- *Benchmark Type.* The type of a benchmark is defined by the part of the system which is stressed during the benchmarking process. Namely, we distinguish between benchmarks that stress the processor (CPU-intensive), the memory (memory-intensive), the system, the disk, the network or the graphical devices. The motivation behind this classification is that it is application-agnostic and may be applied to all kinds of benchmarks.
- *Memory Usage.* This part of the profile provides information about the memory footprint of the benchmark, as well as the memory allocation variations.
- *Stability.* This metric reflects the execution determinism of a benchmark, namely the possible variations of the above metrics across multiple runs.

2.2 Initial Profile Data

The computation of the above metrics needs detailed data about the execution of a benchmark. It needs timing information both about the benchmark's global execution and about its fine-grained operations. It also needs information about the number, the type and the scheduling of the different execution events.

To collect this data, we decide to use system tracing and work with a historical log containing timestamped information about the different execution events. To ensure minimal system intrusion, we propose to use LTTng [4,5][1] which is a *de facto* standard for tracing Linux systems. Indeed, LTTng is capable of tracing hardware counters, the kernel and user-level operations. Hardware counters are available through the profiling *perf* kernel API, while kernel and user operations are traced with *tracepoints*. For the kernel, predefined tracepoints refer to context switches, interruptions, system calls, memory management and I/O.

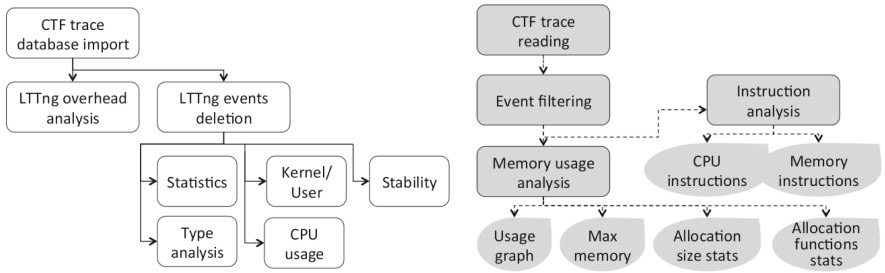
2.3 Profile Computation

The profile computation is a two-phase process which respectively analyzes the kernel and the user-level traces.

The analysis of the kernel trace is implemented as a VisTrails [6] workflow. It thus benefits from its reusability, efficiency and reproducibility features. Indeed, the workflow takes as input a benchmark trace and automatically computes the corresponding profile. The same workflow may be reused for the analysis of a different benchmark or a different configuration of the same benchmark.

The kernel trace analysis workflow follows the standard logic where traces are first captured and stored, and then analyzed offline (Fig. 1a). The first (top) step of the workflow imports the kernel trace into a relational database. The database's generic trace format may represent not only LTTng traces but also other formats [7]. This step also reconstructs processes' states (not active or active and executing a given function).

[1] http://lttng.org.



(a) Kernel Trace Analysis (b) User Trace Analysis

Fig. 1. Profile computation process

The intermediary steps focus on LTTng tracing. The second step characterizes LTTng's tracing overhead in terms of number of traced events and execution slowdown. The third step filters out LTTng-related events to focus only on the benchmark's performances. All computations are done via SQL requests.

The last steps provide execution statistics in terms of number of execution events and execution duration, categorize the execution events to characterize the type of the benchmark, compute the repartition between kernel and user time, analyze the CPU occupation and explore benchmark stability.

For the user-level trace analysis, the trace data is read and streamed to the analysis treatments that operate in parallel and on-the-fly (Fig. 1b). This approach is motivated by the fact that the database-oriented store-and-later-analyze approach does not scale in the case of big execution traces. Indeed, execution traces may easily size up to several GB and their database import and subsequent analysis is costly in terms of storage and computation time.

The user-level trace analysis comprises several modules that process the trace in a pipeline. The first module reads directly the initial trace and transfers it to the events filtering module. The filtering module forwards to the subsequent modules only the information related to the benchmark processes. The memory usage analysis module provides information about the variations in the benchmark's allocated memory, about the size of the allocated chunks of memory and about the frequency of usage of the memory allocation functions. The trace also provides information about the hardware counters which are used to quantify the user-level computations and the memory accesses.

3 Profiling the Phoronix Test Suite

This section details our benchmark profiling in the Phoronix Test Suite case.

3.1 The Phoronix Test Suite

The *Phoronix Test Suite* (*PTS*) [3] provides a set of benchmarks targeting different aspects of a system. *PTS* is available on multiple platforms including Linux, MacOS, Windows, Solaris and BSD.

PTS comes with some 200 open-source test programs. It includes hardware benchmarks typically testing battery consumption, disk performance, processor efficiency or memory consumption. It also targets diverse environments including OpenGL, Apache, compilers, games and many others.

PTS provides little information about benchmarks' logic and internals. Even if each benchmark is tagged as one of *Disk*, *Graphics*, *Memory*, *Network*, *Processor* and *System*, supposedly to indicate which system part is tested, there is no further information on how this tag has been decided or how exactly the benchmark tests this system part.

The repartition of the benchmarks is highly irregular. If we consider that *PTS* benchmarks having the same tag form a benchmark family, the *Network* family contains only one test, while the *Processor* family contains around 80 tests. If, in the first case, a developer has no choice, in the second case, he/she will need to know more about the benchmarks to choose the most relevant.

Table 1. Tagging LTTng Kernel Tracepoints

Family	Events
Processor	`timer_*; hrtimer_*; itimer_*; power_*; irq_*; softirq_*;`
Memory	`kemem_*; mm_*`
System	`workqueue_*; signal_*; sched_*; module_*;rpm_*; lttng_*; rcu_*;`
	`regulator_*; regmap_*;regcache_*; random_*; console_*;gpio_*;`
Graphics	`v4l2_*; snd_*;`
Disk	`scsi_*; jbd2_*; block_*;`
Network	`udp_*; rpc_*; sock_*; skb_*; net_*; netif_*;napi_*;`

3.2 Tracing Phoronix with LTTng

By enabling all LTTng kernel tracepoints, we collect information about scheduling decisions, process management (`exec`, `clone`, `switch` and `exit` function calls) and kernel usage (syscalls). Associated with each traced event is a hardware (CPU) and a software (PID) provenance context.

To analyze which parts of the target system are tested and thus deduce the type of a benchmark, we have analyzed the types of kernel events and mapped them to the *PTS* family tags. For example, the `hmm_page_aloc` and `mm_page_free` are clearly events related to *Memory*-related activity, while `power_cpu_idle` and `htimer_expire` are related to the *Processor*. Table 1 gives the mapping between events and system activity.

User-level tracing is highly dependent of the application to trace and *PTS* benchmarks' are highly heterogenous. To provide a generic tracing solution, we focus on the interface that is commonly used by all benchmarks, namely the standard C library (*libc*). Redefining the `LD_PRELOAD` environment variable and overloading the *libc* functions, it is easy to obtain the information about the memory management functions (`malloc`, `calloc`, `realloc` and `free`) needed for the computation of benchmarks' memory profiles.

Another aspect we are interested in is to characterize the user level behavior of a benchmark in terms of CPU or memory-related activity. To do so, we use the information provided by hardware performance counters. In particular we use the `Instruction` counter, which gives the total number of instructions executed. We also use the `L1-dcache-loads` and `L1-dcache-stores` counters that provide the total number of L1 cache reads and L1 cache writes. As all data access go through the L1 cache, the sum of those two values gives the total number of data related instructions. To get the number of computation related instructions, we use the difference `Instruction` − (`L1-dcache-stores` + `L1-dcache-stores`).

3.3 Experimental Setup

We have worked with 10 *PTS* bechmarks, namely `compressgzip`, `ffmpeg`, `scimark2`, `stream`, `ramspeed`, `idle`, `phpbench`, `pybench`, `network-loopback` and `dbench`. The set is part of the *PTS recommended* benchmarks which are the most popular ones as determined by the number of downloads and available results [8]. We have used three benchmarks from the *Processor* family, three from the *System* family, two from *Memory*, one *Disk* and one *Network* benchmark.

Each benchmark is run with its default options as defined by the *PTS* system except for the number of runs. Instead of 3 times we run benchmarks 32 times to ensure statistically reliable results [9]. The score for each benchmark, which is benchmark-specific, is computed as the mean value of the 32 obtained scores.

The experiments have been run on three different platforms which helped validate the fact that benchmarks have similar executions, hence profiles, whatever the platform. We have used one UDOO board[2], one Juno board[3] and a desktop machine. In the following we show results from the UDOO and the Juno boards. The UDOO has an i.MX 6 4core ARM CPU at 1 GHz, a Cortex-M3 coprocessor and 1 GB of RAM. It runs the multi-platform Debian kernel for ARM `armmp3.16`.The Juno board has one 2core CortexA57 and one 4core CortexA53 processors with 2 GB of RAM. It runs a Debian kernel (`4.3.0-1-arm64`).

3.4 LTTng Overhead and Benchmark Stability

Our analysis starts with an evaluation of LTTng's perturbation of the target system. In terms of execution duration, both for kernel and user traces, LTTng's overhead is negligible as it is less that 1 %. The only notable exception is the case of the `phpbench` benchmark slowed down by 78 % by user-level tracing because of its heavy use of memory operations.

In terms of collected events, LTTng-related events account for 10 % to 26 % in kernel traces and between $100K$ and $200K$ in user traces. To prevent bias in statistics metrics computations, these events are filtered out and ignored during trace analysis. Finally, considering benchmarks' results, scores from executions with and without tracing do not differ more than 2.5 %.

[2] http://www.udoo.org.
[3] http://www.arm.com/products/tools/development-boards/.

Table 2. Information on Phoronix Benchmarks (UDOO board)

Benchmark	Exec.(m)	Size(GB)	Idle	SD	User / Kernel ratio
compress-gzip	94	5.18	76	0.03	78% 22%
ffmpeg	221.90	62	62	0.01	99% 01%
scimark2	22.41	0.28	76	0.00	99% 01%
stream	7.00	0.35	33	0.01	99% 01%
ramspeed	2019.25	30.20	24	0.00	83% 17%
idle	1.09	0.01	99	0.60	48% 52%
phpbench	649.72	15.35	75	0.00	97% 03%
pybench	267.71	1.60	75	0.00	99% 01%
dbench	915.72	58.20	0	0.03	01% 99%
network-loopback	90.92	25.50	54	0.00	11% 89%

Table 2 summarizes information about the 32 runs of the considered benchmarks. Namely we have the global execution time, the corresponding trace size, the relative time spent in idle mode (`idle` stands out), the standard deviation for benchmarks' duration and the ratio between user and kernel time. There are important differences, even between benchmarks belonging to the same family. A simple recommendation to developers would be to use shorter benchmarks.

We have investigated benchmark stability over the 32 runs. Having reverse-engineered the Phoronix launching process and identified the trace parts about the 32 distinct runs, we evaluated the stability of the considered profile metrics (number of events, execution time, kernel and user time, number of cores). Considering the benchmark execution time, for example, we have computed the mean value, the maximum, the minimum and the standard deviation. The latter is close to zero meaning that the benchmarks are stable. The only exception is `idle` whose variation may be explained by its short execution time (6 ms). The analysis of the other parameters shows similar results.

3.5 Benchmark Types

A first simple classification of Phoronix benchmark is to consider the ratio of kernel versus user operation. Table 2 gives this ratio and shows that there are only 4 benchmarks spending significant time in kernel mode. It is worth noting that the ratio here is computed over benchmarks' useful execution time and ignoring the idle time. For `idle`, for example, this represents only 1 % of its total execution time. We can conclude that the others are either CPU- or memory intensive.

If we use the classification of kernel events introduced in Sect. 3.2 and use the number of traced events, we obtain the kernel profiles shown in Fig. 2. We can clearly see that there is no benchmark testing the graphics subsystem (no graphics events) and that `network-loopback` and `dbench` respectively test the network and the disk. Indeed, they are the only ones with a significant amount of respectively *Network* and *Disk* events.

If we consider the benchmarks tagged as *Memory* within Phoronix, `stream` has an important kernel activity and its kernel profile confirms the frequent usage

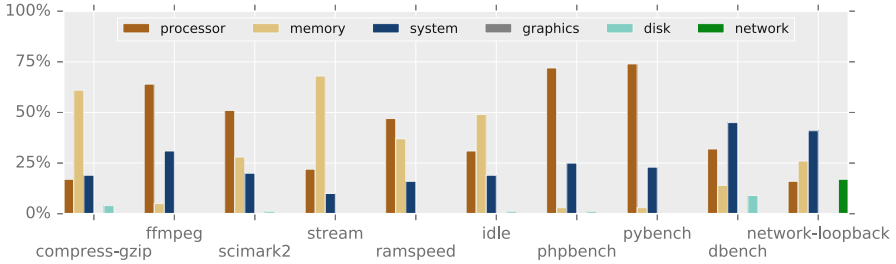


Fig. 2. Kernel Operation (UDOO board)

of memory-related functions. However, the profile of `scimark2` is different and to verify whether the benchmark is indeed memory-intensive one should consider its execution in user mode.

In the *Processor* family, `ffmpeg` and `scimark2` have the expected kernel profiles with predominant *Processor* events. `compress-gzip`, however, shows an important memory management activity so the profile computation should consider the user-level information.

Our analysis of the *System* Phoronix family made us understand that it includes various benchmarks testing different software systems (or layers, or middleware) and it does not necessarily focus on the operating system level. `idle` does test the operating system and quantifies the execution time of a program doing nothing. However, `phpbench` and `pybench`, which, by the way, have similar kernel profiles, respectively test the performances of PHP and Python code.

3.6 CPU Usage and Parallelization

An interesting aspect we have investigated is the way benchmarks use the available processors. In the case of the UDOO platform, we can see that benchmarks have quite different parallelization schemes (Fig. 3). The `idle` benchmark does not use the CPU, as expected. The `pybench` benchmark uses only 3 CPU out of 4. The other benchmarks do use the 4 processors but only `dbench`, `stream`

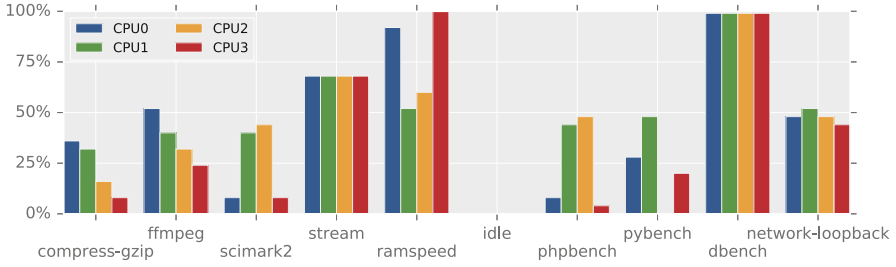


Fig. 3. CPU Usage and Parallelization (UDOO board)

and `network-loppback` are totally balanced. `dbench` uses the CPUs at 100 %, `stream` at 68 % and `network-loppback` 45 %.

Another interesting observation is that there are two couples with quite similar CPU usage profiles. These are `scimark2` and `phpbench`, on one hand, and `compress-gzip` and `ffmpeg`, on the other. However `scimark2` and `phpbench` belong to the *Processor* and *System* family respectively. As for `compress-gzip` and `ffmpeg`, the two being of the *Processor* family, it may be better to consider the `ffmpeg` benchmark which runs longer but makes a more efficient usage of the platform processors.

Table 3. Maximum Memory and Instructions repartition (Juno board)

Benchmark	Memory (KB)	Instructions repartition (Compute / Memory)
compress-gzip	138	72% / 28%
ffmpeg	5 210	71% / 29%
scimark2	16 779	64% / 36%
stream	9	58% / 42%
ramspeed	3 456 108	49% / 51%
idle	2	56% / 44%
phpbench	3 225	54% / 46%
pybench	1 827	56% / 44%
dbench	225 608	77% / 23%
network-loopback	1 123	55% / 45%

3.7 Memory Usage Profile

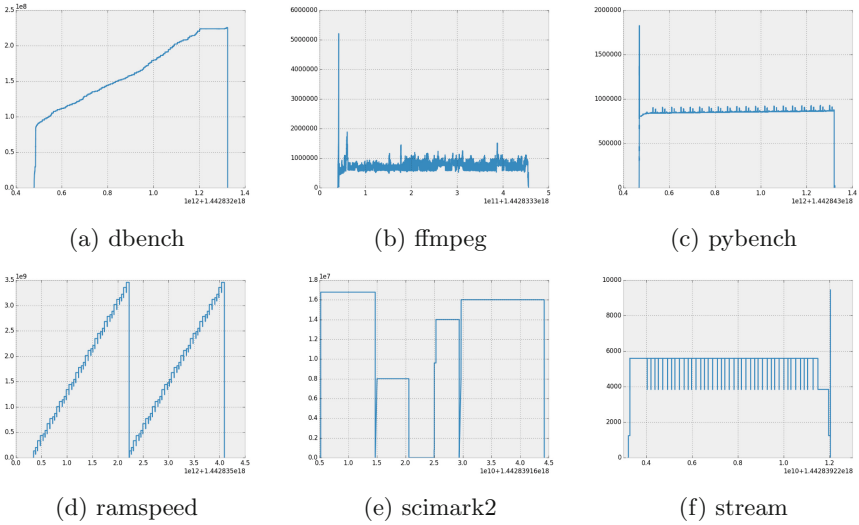
User-level tracing yields interesting information about the differences between benchmarks. The maximum memory allocated by benchmarks, for example, is quite varying (Table 3). Indeed, only `ramspeed` with 3.45 GB uses almost all virtual memory available on the UDOO board (4 s). `stream` and `idle` are memory light-weight as they consume respectively 9 KB and 2 KB.

The results from profiling of user-level operations and classifying them into computational and memory-related are given in the third row of Table 3. These confirm that the *Memory* benchmarks, `ramspeed` and `stream` do use intensively the memory as expected. The other benchmarks which reveal to be memory-intensive are the *System* ones we have selected, namely `phpbench` and `pybench`. As for `idle` and `network-loopback`, these are not representative as the time spend in user mode is very short (0.6 % of the total execution time for `idle` and 5 % for `network-loopback`).

Figure 4 gives the evolution of the memory usage of benchmarks over time and shows that there are various behaviors.

4 Related Work

Current benchmark-oriented efforts [3,10,11] focus on the problems of providing a set of benchmarks which is to be *complete*, *portable* and *easy* to use. Indeed, the



(a) dbench (b) ffmpeg (c) pybench

(d) ramspeed (e) scimark2 (f) stream

Fig. 4. Memory usage (bytes) over time (ns). (UDOO board)

goal is to provide benchmarks that cover different performance aspects, support different platforms and can be automatically downloaded, installed and executed. However, the classification of benchmarks is *ad hoc* and there is no detailed information about their functionality. Our proposal is a step forward as it allows benchmark comparison through automatic profiling.

Our work is motivated by the need of identifying relevant and representative benchmarks and is thus related to [12]. The authors use simulation to obtain source-code-related statistics which are further treated with cluster-based methods to identify a minimal representative benchmark set. Our solution is complementary as it considers real-world settings, applies to binary programs and, most of all, automates the analysis. It can benefit from the learning techniques to identify the representative set of Phoronix benchmarks.

Our proposal can be seen as a profiling tool for benchmarks. However, existing profiling tools [13–16] typically provide detailed low-level information on a particular system aspect. Moreover, they are system dependent. Our proposal is applicable to all types of benchmarks, on different platforms and provides a macroscopic vision of their behavior.

The major aspects of our profile computation are trace analysis and workflow management. Concerning trace analysis, most existing tools are system and format specific [17,18] and limit themselves to time-chart visualizations and basic statistics. In our work, trace analysis is brought to a higher level of abstraction. It is based on generic data representation of traces and thus may be applied to execution traces (hence benchmarks) from different systems. It is not organized as a set of predefined and thus limited treatments but may be configured and enriched to better respond to the user needs. Finally, its structuration in

terms of a deterministic workflow allows for automation and reproducibility of the analysis process.

As for workflow-oriented tools [19,20], they focus on formal specification, automation, optimisation and reproducibility of computations aspects. Generic trace analysis and especially the problem of huge traces have not been considered.

5 Conclusion and Ongoing Work

We have presented in this paper a workflow-based tool for automatic profiling of benchmarks. The result is a unified profile which characterizes a benchmark, allows the comparison among benchmarks and thus facilitates the choices of a system performance analyst. We have illustrated our approach with the Phoronix Test Suite and have experimented with embedded and desktop Linux-based platforms. We have successfully produced the profiles for several benchmarks exhibiting their different characteristics. Our experimentation puts forward the fact that the initially provided description is far from sufficient for understanding the way benchmarks test the target system.

In our work we have taken advantage of workflows' useful features such as automation, result caching and reproducibility. However, most workflow systems do not properly address the data management issues when it comes to manipulating big data sets. In this regard, we have shown that workflow tools should provide for pipelining, streaming and parallel computations. An ongoing collaboration with the VisTrails team brings these features to the VisTrails tool.

The benchmark profiles we provide are easy to compute, to understand, and to compare. All metrics observed, tools used, and profiles drawn, do not depend on any specifics of the benchmarks.

A long term research objective would be to provide generic means for reflecting the benchmark specifics into the profile and thus help even more the performance evaluation work of an analysis.

References

1. Almeida, J.B., Frade, M.J., Pinto, J.S., de Sousa, S.M.: Rigorous Software Development: An Introduction to Program Verification. Springer, London (2011)
2. A Practitioner's Guide. Cambridge University Press, New York (2000)
3. Phoronix Test Suite. http://www.phoronix-test-suite.com/
4. Desnoyers, M.: Low-Impact Operating System Tracing. PhD thesis (2009)
5. Chen, K.Y., Chang, Y.H., Liao, P.S., Yew, P.C., Cheng, S.W., Chen, T.F.: Selective Profiling for OS scalability study on multicore systems. In: IEEE 6th International Conference on Service-Oriented Computing and Applications, pp. 174–181 (2013)
6. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C., Silva, C.T., Vo, H.T.: VisTrails: visualization meets data management. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 745–747 (2006)
7. Pagano, G., et al.: Trace management and analysis for embedded systems. In: IEEE 7th International Symposium on Embedded Multicore Socs, September 2013

8. Phoronix Test Suite, User Manual. http://www.phoronix-test-suite.com/documentation/
9. Jain, R.: The Art of Computer Systems Performance Analysis (1991)
10. Industry-Standard Benchmarks for Embedded Systems. http://www.eembc.org/benchmark/products.php/
11. Future Benchmarks and Performance Tests. http://www.futuremark.com/
12. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. IEEE Trans. Comput. **55**(6), 769–782 (2006)
13. Yamamoto, M., Ono, M., Nakashima, K., Hirai, A.: Unified performance profiling of an entire virtualized environment. In: Second International Symposium on Computing and Networking (CANDAR), pp. 106–115, December 2014
14. Gouigoux, J.-P.: Practical Performance Profiling: Improving the Efficiency of .NET Code. Red gate books, United Kingdom (2012)
15. Seward, J., Nethercote, N., Weidendorfer, J.: Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications. Network Theory Ltd., Bristol (2008)
16. Janjusic, T., Kartsaklis, C.: Glprof: a Gprof inspired, callgraph-oriented per-object disseminating memory access multi-cache profiler. In: Procedia Computer Science, International Conference on Computational Science, ICCS Computational Science at the Gates of Nature, vol. 51, pp. 1363–1372 (2015)
17. Knüpfer, A., et al.: Score-p: a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In: Tools for High Performance Computing 2011. ZIH, Dresden, pp. 79–91, September 2011
18. Prada-Rojas, C., Santana, M., De-Paoli, S., Raynaud, X.: Summarizing embedded execution traces through a compact view. In: Conference on System Software, SoC and Silicon Debug S4D (2010)
19. Cohen-Boulakia, S., Leser, U.: Search, adapt and reuse: the future of scientific workflows. SIGMOD Records **40**(2), 1187–1189 (2011)
20. Qin, J., Fahringer, T.: Scientific Workflows, Programming, Optimization, and Synthesis with ASKALON and AWDL. Springer, Heidelberg (2012). ISBN 978-3-642-30714-0