Regenerating and Quantifying Quality of Benchmarking Data Using Static and Dynamic Provenance

Devarshi Ghoshal[1(✉)], Arun Chauhan[1,2], and Beth Plale[1]

[1] School of Informatics and Computing, Indiana University, Bloomington, IN, USA
{dghoshal,achauhan,plale}@cs.indiana.edu
[2] Google Inc., Mountain View, CA, USA

Abstract. Application benchmarks are critical to establishing the performance of a new system or library. But benchmarking a system can be tricky and reproducing a benchmark result even trickier. Provenance can help. Referencing benchmarks and their results on similar platforms for collective comparison and evaluation requires capturing provenance related to the process of benchmark execution, programs involved and results generated. In this paper we define a formal model of benchmark applications and required provenance, describe an implementation of the model that employs compile time (static) and runtime provenance capture, and quantify data quality in the context of benchmarks. Our results show that through a mix of compile time and runtime provenance capture, we can enable higher quality benchmark regeneration.

1 Introduction

Application benchmarks are an important way to establish the speed of a new system or library. But benchmarking a system can be tricky and reproducing a benchmark even trickier; a single compile time parameter can give vastly different results depending on whether it is set or not. Analyzing and recording benchmark results is a manual task that depends on the knowledge of the evaluator. The manual nature increases the possibility of missing information, incorrectly logged results and skipped parameters and configurations that affect the program behavior. All these factors affect the quality of the results generated by the benchmark process.

The metadata required to assess the quality of benchmark results to reproduce program behavior and quality of the evaluation process can be complex and expensive. Provenance is a type of metadata used to capture the lineage of data. Provenance from benchmark executions can be used to describe the lineage of benchmark results and the evaluation process involved. It helps in understanding the quality and enables regeneration and referencing of benchmarks. Essentially provenance traces for benchmark results account for the following – (a) ensuring benchmarks were executed correctly, (b) understanding the set of parameters and configurations used for generating the results and (c) keeping track of the benchmarks and their results for evaluation.

B. Ludäscher and B. Plale (Eds.): IPAW 2014, LNCS 8628, pp. 56–67, 2015.
DOI: 10.1007/978-3-319-16462-5_5

For standard application benchmarks, guaranteeing the correct execution of benchmarks and keeping track of the results require intelligent provenance capturing techniques. Log based provenance capture mechanisms [GP13] can be used for collecting provenance from benchmarking. But it is important to guarantee that neither the results are tampered with nor important factors affecting the results ignored. Existing provenance management frameworks [CCPR13, GT12] manage provenance at the application granularity. For benchmarking data, provenance needs to be managed for both the application (benchmark) and the target system (hardware or software or both) for the benchmark. Additionally, existing provenance capturing techniques often require modifications to the filesystem [GT12,MRHBS06], application specific program instrumentations [CCPR13,ABGK13] and/or trapping system calls [GT12] which are not viable due to varied nature of benchmarks and high-degree of system perturbations.

In this paper we identify essential characteristics of provenance in benchmarking and propose a formal model of provenance from application benchmarks. We describe a framework based on the provenance model that captures provenance from application benchmarks both statically at compile time and at runtime in order to validate, regenerate and reference results for future research. This paper makes the following contributions:

- a formal model of benchmark applications and required provenance
- an implementation of the model that employs compile time (static) and runtime (dynamic) provenance capture
- quantification of data quality in the context of benchmarks
- a PROV representation of the data model for provenance of benchmarking applications.

The remainder of the paper is organized as follows. In Sect. 2 we discuss related work. Section 3 proposes a formal model of provenance capture from application benchmarks. Section 4 describes our methodology and the implementation for identifying and capturing provenance using the provenance capture model. We evaluate our model and framework in Sect. 5. Finally, we present our conclusions in Sect. 6.

2 Related Work

Use-cases of provenance. Provenance capture, representation and use has been studied for e-science workflows [Mea05], file systems [MRHBS06], semantic web [CBHS05] and databases [CCT09]. The use of provenance in determining the quality of scientific data and data provenance has also been shown [SP11]. Provenance from scientific executable document systems [Yea12] are also implemeted. But using provenance for quantitative and qualitative analysis of benchmarking results has not been studied earlier.

Models of provenance capture. Several models have been proposed to identify and capture provenance [CAA07]. Bower et al. [BML12] proposes a dependency

rule language for capturing fine-grained provenance from workflow traces but requires user-defined rules and runtime traces. We mostly rely on static analysis of source code for fine-grained provenance.

Provenance identification and capturing mechanisms. Provenance-aware solutions [MRHBS06], and language extensions [CAA07] for provenance identification have been proposed. Provenance capture by analyzing audit logs and semi-automated code instrumentation [GT12,CCPR13] have also been developed. We developed output-monitoring and compiler-driven provenance identification mechanisms for collecting provenance from application benchmarks.

Quality assessment. Quality of provenance data and using provenance for understanding the quality of data [CP12,HZ09] are important aspects of quality measurement in provenance. But very little or no work has been done to quantify data and provenance quality at the system level. In our work we quantify the quality of benchmark result, which is a provenance artifact, based on the level of intrusion through external factors.

3 Formalization of Benchmark Provenance

In this section we provide a formal representation of a benchmarking application and use that to define provenance capture.

3.1 Model of Benchmarking

A benchmark application has specific properties where a property is a pair $(n : v)$ where n is the name of the property and v is the corresponding value. A property can either be a static characteristic of the program (or set of programs that build the application) or a dynamic value only known at runtime. If we consider M execution instances of an application with N distinct properties, we can define two categories of properties as follows:

Variants: A set of properties that changes or may change for an execution instance, $i \in M$, of the application. Hence for a particular variant property, its value varies with i. The *variant* set is then defined as,

$$\forall i \in M, Variant(i) = \{(n : v_i) \mid \exists\ n \in N,\ s.t.\ v_i = f(i, n)\} \tag{1}$$

Since benchmarks are executed multiple times, results are concluded by aggregating individual output from each instance of the benchmark execution. Hence, it is important to unify the variants from all benchmark execution instances in order to preserve the provenance of the final output. Unifying variants over all execution instances $i = (1, \ldots, M)$ gives the total set of variants as,

$$\begin{aligned} \mathbb{V} &= \bigcup_{i=1}^{M} Variant(i) \\ &= \bigcup_{i=1}^{M} \{(n : v_i) \mid \exists\ n \in N,\ s.t.\ v_i = f(i, n)\} \end{aligned} \tag{2}$$

Practically, variants for application benchmarks consist of resource usage like CPU load at the time of benchmark execution and available memory, configuration parameters etc.

Invariants: A set of properties that remains constant over multiple execution instances of an application. The value of an invariant is only dependent on the name of the property. Invariants are, therefore, defined as,

$$\forall i \in M,\ Invariant(i) = \{(n : v) |\ \exists\ n \in N,\ s.t.\ v = g(n)\} \tag{3}$$

Since invariants are independent of the execution instance i, unification results in distinct (n:v) pairs of an application benchmark as,

$$\begin{aligned} \mathbb{I} &= \bigcup_{i=1}^{M} Invariant(i) \\ &= \bigcup_{i=1}^{M} \{(n : v) |\ \exists\ n \in N,\ s.t.\ v = g(n)\} \end{aligned} \tag{4}$$

Examples include names of the programs, associated libraries, create-date of the benchmark binary, hostname(s) etc.

Since a benchmark is used for evaluating a system, where we define a system S as a software or hardware entity that has certain properties, we can define benchmark as a partial function,

$$\beta : (\mathcal{I}_\beta, \mathbb{V}, \mathbb{I}, S) \mapsto \mathcal{R}_\beta \tag{5}$$

where $\mathcal{I}_\beta$ is the set of input-data, $\mathbb{V}$ is the set of variants, $\mathbb{I}$ is the set of invariants, S is the evaluated system and $\mathcal{R}_\beta$ is the set of output results.

To summarize, a benchmark with a set of properties $\mathbb{V}$ and $\mathbb{I}$, evaluates a system S generating the result-set $\mathcal{R}_\beta$ for an input-set $\mathcal{I}_\beta$. It is a partial function because invariants do not map to the result-set but are properties that are unique to the benchmark.

3.2 Model of Provenance Capture

We base our model for provenance on the model of benchmarking defined above. We make no assumptions about the equivalence of inputs and outputs of a benchmark and that collected by our model of provenance capture. So, we denote the output result-set collected by our model of provenance as $\mathcal{R}_\mathcal{P}$. Similarly, we also consider the input data-set collected by the model of provenance as $\mathcal{I}_\mathcal{P}$.

Static Provenance. We define static provenance capture as a function that maps a benchmark to its invariants.

$$\delta : \beta \mapsto \mathbb{I} \tag{6}$$

Any property that does not vary with different execution instances of a benchmark program but identifies it uniquely is considered during static provenance capture. Hence, artifacts for static provenance capture can be determined statically without executing the benchmark.

Runtime Provenance. Runtime provenance capture, on the other hand, captures provenance information for every execution instance of a benchmark. It depends on the runtime characteristics and parameters of benchmark execution. We define runtime provenance capture as a function that maps a set of results to a set of inputs, corresponding benchmark and variants.

$$\gamma : \mathcal{R}_{\mathcal{P}} \mapsto (\beta, \mathcal{I}_{\mathcal{P}}, \mathbb{V}) \tag{7}$$

All data-items that affect the benchmark results but can only be determined during benchmark execution are captured during runtime provenance capture.

3.3 Quantification of Data Quality

Since no assumptions are made about the equivalence of inputs and outputs of a benchmark and the captured provenance, there may be discrepancies between the published inputs and outputs of a benchmark and that collected through provenance capture. In the ideal situation, $\mathcal{R}_{\mathcal{P}} \equiv \mathcal{R}_{\beta}$ and $\mathcal{I}_{\mathcal{P}} \equiv \mathcal{I}_{\beta}$.

Trust. We define $\parallel \mathcal{R}_{\mathcal{P}} - \mathcal{R}_{\beta} \parallel$ to denote the quantitative difference between the results, i.e., the number of results that differ in the two sets. Similarly, $\parallel \mathcal{I}_{\mathcal{P}} - \mathcal{I}_{\beta} \parallel$ denotes the quantitative difference between the inputs. For a set of invariants, $\mathbb{I}$ and variants, $\mathbb{V}$ of a benchmark, β the trust, $\mathbb{T}$ of the result data-set is then measured by the following equation:

$$\mathbb{T} = \left(1 - \frac{\parallel \mathcal{R}_{\mathcal{P}} - \mathcal{R}_{\beta} \parallel}{max(\mid \mathcal{R}_{\mathcal{P}} \mid, \mid \mathcal{R}_{\beta} \mid)}\right)\left(1 - \frac{\parallel \mathcal{I}_{\mathcal{P}} - \mathcal{I}_{\beta} \parallel}{max(\mid \mathcal{I}_{\mathcal{P}} \mid, \mid \mathcal{I}_{\beta} \mid)}\right) \tag{8}$$

where,
$\parallel X - Y \parallel$ returns the count of mutually exclusive elements of X and Y,
$\mid X \mid$ is the cardinality of a set X.

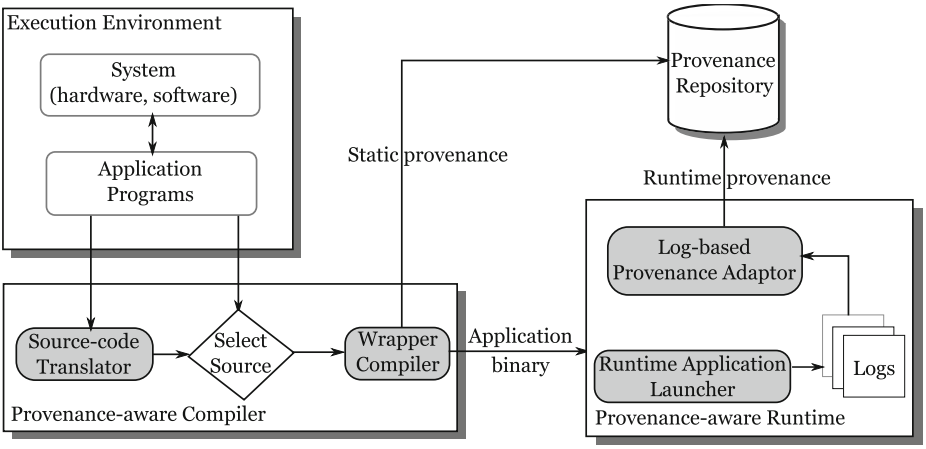


Fig. 1. Framework for capturing provenance for benchmarking data.

In other words, if the input and result generated by a benchmark differs from what its provenance says, then the data is not trustworthy. For simplicity, we consider each input and result to be of equal importance. We also assume that the values of invariants and variants are always within the range of the values captured through provenance. A direct result that follows through the above quantification is the measure of *reproducibility* which is a property of the benchmark result and is a boolean value that determines if a benchmark result is reproducible or not. It is defined in terms of 'trust'.

Definition: Given a set of invariants $\mathbb{I}$ *and variants* $\mathbb{V}$*, a result-set* $\mathcal{R}_\beta$ *is reproducible for a benchmark* β *iff* $\mathbb{T} = 1$.

4 Provenance-Aware Benchmarking Framework

The formal model is the foundation upon which is built the framework for capturing provenance from application benchmarks as shown in Fig. 1. The framework has two pieces: a static capture component that is built into compile time activity. Run-time is also made provenance-aware through runtime capture.

4.1 Static Provenance Capture

We propose a provenance-aware compiler for static capture of provenance. The compiler is implemented as a wrapper over standard gcc or icc compilers. To enable provenance-aware compilation, a user replaces all calls to the corresponding compiler by call to the wrapper compiler provcc which captures 'invariants' as provenance elements during program compilation. Essentially,

$$\mathsf{provcc} : \beta \mapsto \mathbb{I}$$

where, β is a benchmark and $\mathbb{I}$ is a set of invariants.

4.2 Runtime Provenance Capture

The runtime provenance capture is divided into two modules – (a) provenance-aware runtime and (b) provenance adaptor.

Provenance-Aware Runtime. The second piece of the solution, a provenance-aware runtime, executes and captures provenance information including the variants, inputs and results for a benchmark. All benchmarks are executed via the runtime application launcher provrun which can capture the results from both stdout and files. From our provenance model, provrun is the mapping function δ.

$$\mathsf{provrun} : \beta \mapsto (\mathcal{I}_\mathcal{P}, \mathbb{V}, \mathcal{R}_\mathcal{P})$$

where, β is a benchmark, $\mathcal{I}_\mathcal{P}$ is the set of inputs captured by provrun, $\mathbb{V}$ is a set of variants, $\mathcal{R}_\mathcal{P}$ is the set of output results captured by provrun.

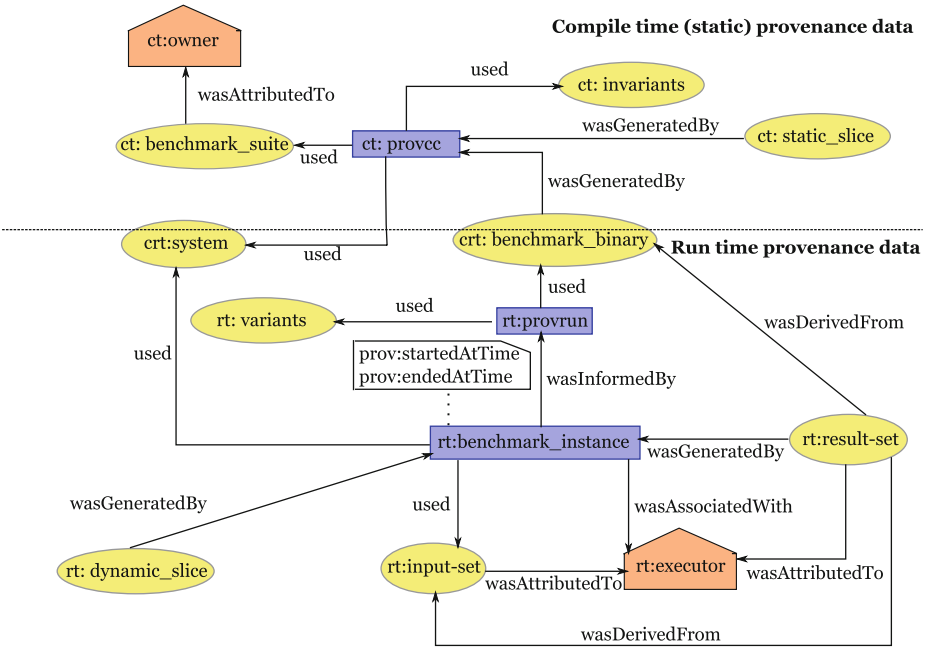


Fig. 2. PROV model for capturing provenance from application benchmarks.

Provenance Adaptor. The second phase of the runtime provenance capture collects, combines and translates provenance information captured in log files into a single provenance graph. It shows the lineage of benchmark results for all execution instances of a benchmark on a system. The complete provenance graph is generated by combining the provenance information collected statically during compilation and dynamically by the runtime system.

4.3 Fine-Grained Provenance Capture

The compiler wrapper is augmented with an additional source-to-source translator module that allows for source-code instrumentation for fine-grained provenance capture. This module allows users to automatically identify and mark regions in the code to generate provenance information. We developed the module using the ROSE compiler framework. It builds a system dependency graph for the benchmark programs and marks regions of the code based on the granularity of provenance information. This module is responsible for generating two slices of the benchmark program: a static slice created during compilation and a dynamic slice generated during benchmark execution. Static slice is used for deriving the mapping between inputs and outputs and preserving the interprocedural dependencies. Whereas the dynamic slice is used to capture the actual parameters passed across the functions for generating the output.

4.4 PROV Model for Benchmark Provenance

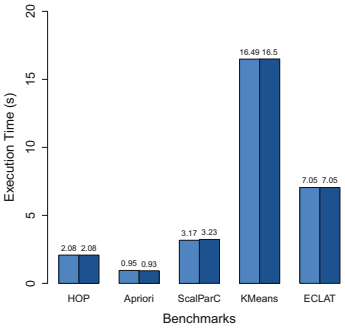
Figure 2 shows the PROV data model for application benchmarks. Based on the formal model of provenance capture, the PROV model shows compile-time and run-time provenance capture from application benchmarks. The owner is a PROV agent who is responsible for creating a benchmarking suite. The executor, on the other hand, is a PROV agent who executes a benchmark or evaluates a system using a benchmark. Each benchmark execution has an associated time attribute that captures the start and end time of execution. Invariants and variants are captured as part of the provenance by the provenance-aware compiler and the runtime respectively. Fine-grained provenance is captured by generating the static and dynamic slices of a benchmark program. Finally, the output result is derivation from the benchmark binary and the input data-set which are used to evaluate a system.

5 Evaluation

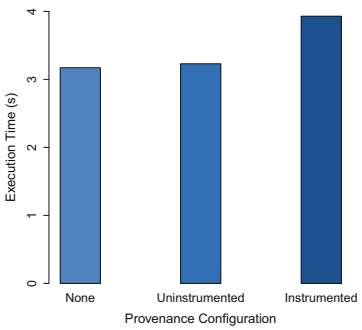
We experimentally evaluate our methodology using six benchmarks from the NU-MineBench [NOZ+06] benchmarking suite and analyze both overheads and significance of provenance capture from benchmarks. NU-MineBench contains a mix of several representative data mining applications from different application domains. It is used for computer architecture research, systems research, performance evaluation, and high-performance computing. The applications used are: **HOP** – a density-based data clustering. **Apriori** – association rule mining. **ScalParC** – decision-tree based data classification. **K-means** – (and Fuzzy K-means) for data clustering. **ECLAT** – association rule mining. **Semphy** – structure learning algorithm that is based on phylogenetic trees.

Tests were run on a quad-socket, 8-core (32 total cores) AMD Opteron system with 512 GB of memory running 64-bit Red Hat Enterprise Linux. For evaluating the runtime overhead, benchmarks are executed 10 times. As a micro-benchmark, we measure runtime overhead and for higher quality benchmark regeneration, we evaluate the model and the framework along three dimensions – (a) computing if the result is reproducible (quality quantification), b) what is required to regenerate the result (reproducibility data) and (c) how can the result be regenerated (reproducibility steps).

Runtime Overhead. Executing the benchmarks through the provenance-aware framework shows no or very little overheads as shown in Fig. 3a. This is because the benchmark execution is completely uninterrupted and provenance information is logged only in two stages – (a) prior to the execution and (b) when the execution completes. However, we also capture fine-grained provenance by running an instrumented version of the benchmark. In order to enrich provenance information, benchmark programs are marked at specific regions during compilation by the provenance-aware compiler. For this evaluation we only mark function calls which tracks inter-procedural data flow in order to derive the exact mapping between inputs and outputs. This instrumentation results in relatively high overheads as shown in Fig. 3b.



(a) Benchmark execution through provenance-aware runtime



(b) Benchmark execution with different granularity of provenance capture for ScalParC

Fig. 3. Performance overhead in benchmark execution through provenance-aware framework. There are very little or no overheads for provenance capture without instrumenting the benchmark. However, fine-grained provenance capture through source-code instrumentation starts incurring higher overheads.

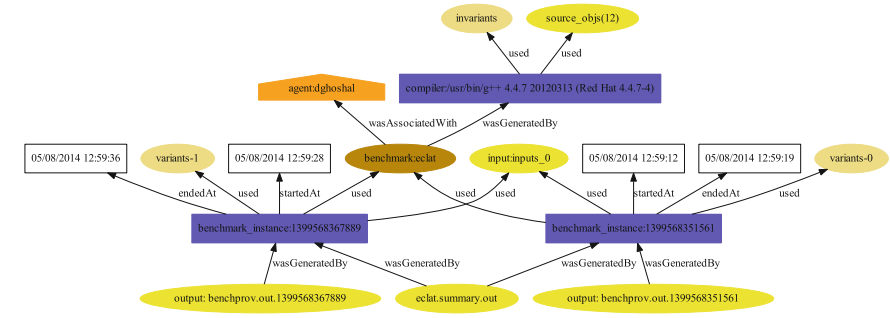
Table 1. Quality assessment derived from provenance

	HOP	Apriori	ScalParC	K-Means	ECLAT
Num results	1	3	5	45	12
$\parallel \mathcal{R}_{\mathcal{P}} - \mathcal{R}_{\beta} \parallel$	0	1	2	0	1
Trust value	1.00	0.67	0.60	1.00	0.92

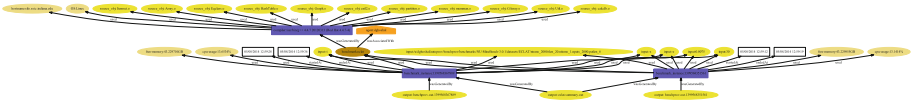
Table 2. Elements of compile time (static) provenance as captured by provcc

Benchmark	Srcs	Objs	Compilation-flags	Opt-flags	Linker-library
HOP	6	6	-fopenmp -Wno-write-strings	-O	libm
Apriori	5	5	-fopenmp -DBALT	-O2	libm
ScalParC	4	4	-fopenmp	-O2	libm
K-Means	4	4	-fopenmp	-O2	libm
ECLAT	14	11	-Wno-non-template-friend	-O3	libm, libc
SEMPHY	23	15	-Wall -Wno-sign-compare -DLOG	-O3	../../lib
			-ftemplate-depth-32		libSEMPHY.a
					libEvolTree.a

Quality Quantification. We calculate trust values for different benchmarks by introducing discrepancies in the result data-set by introducing errors as shown in Table 1. These errors are either system or human errors of reporting results, missing information, inconsistent values of variants etc. For example, for the



(a) Aggregated provenance graph



(b) Expanded provenance graph

Fig. 4. Provenance graphs for ECLAT - (a) shows the aggregated view of the original provenance graph as shown in (b) for 2 runs of the ECLAT benchmark.

ECLAT benchmark run, if we change the value of the `support` parameter from 0.0075 that is captured through provenance to 0.0080, we are unable to reproduce the output result as predicted through our reproducibility metric. This is because the trust value, $\mathbb{T}$ is less than 1. For benchmark results, **trust value** $= 1$ iff $\| \mathcal{R}_{\mathcal{P}} - \mathcal{R}_{\beta} \| = 0$. In other words, a result can only be trusted and hence reproduced, when the benchmark and its provenance points to the same result-set.

Regeneration Using Provenance. Provenance for regenerating benchmarking data can be categorized into two phases based on our capture model – (a) information useful for regenerating the application benchmark and (b) information for regenerating the benchmark results. For regenerating the benchmark application, we capture its provenance that includes the compilation flags, platforms, source programs (invariants) etc. Table 2 shows a list of provenance elements that are captured using the provenance-aware compiler. For regenerating the benchmark results, associated runtime characteristics (variants) and the input data along with the provenance of the application benchmark are captured. In case of fine-grained provenance, the detailed mapping between inputs and outputs and interprocedural dataflow are also captured.

Since benchmark results are most often an aggregation of individual runs of a benchmark, the correlation between individual results and configurations are important to record as part of the provenance. A connected provenance graph shows the importance of recording, correlating and linking individual provenance traces of a benchmark. As shown in Fig. 4 the aggregated results for ECLAT are written to a single output file through different runs of the benchmark.

So, the summary output file is a result of all the inputs, compiled binaries and configurations of the benchmark over a set of multiple runs. Generating and representing a complete provenance graph describing the steps and data for multiple instances of the benchmark, is useful for understanding and regenerating the result.

6 Conclusion and Future Work

There are several open questions remaining. The model and framework captures provenance from benchmarks running in non-distributed environments. Distributed environments pose challenges in correlating benchmark results, tracing failures, and input-output mapping. Too, the equation we pose for calculating trust is binary. It captures perfect reproducibility but does not allow epsilons of change in the execution trace or static analysis that do not compromise trust. In the case where benchmarks are run in distributed environments, acceptable differences like out of order messages or slightly mismatched clocks may occur. Additionally, the work assumes the provenance captured has not been intentionally altered. Our approach assumes availability of benchmark source-code. In the absence of source-code or provenance-aware compilers, special techniques should be developed for identifying and correlating provenance from benchmarking applications transparently without user intervention. The amount and granularity of fine-grained provenance sufficient for validating benchmark execution also needs further research. Finally, post-processing can be done by mining the result-set and associated provenance in order to automatically derive conclusions about the evaluation process and system's performance.

Acknowledgements. This work is funded in part by the National Science Foundation OCI 1148359.

References

[ABGK13] Alper, P., Belhajjame, K., Goble, C.A., Karagoz, P.: Enhancing and abstracting scientific workflow provenance for data publishing. In: The Joint EDBT/ICDT 2013 Workshops, New York, NY, USA, pp. 313–318 (2013)

[BML12] Bowers, S., McPhillips, T., Ludäscher, B.: Declarative rules for Inferring fine-grained data provenance from scientific workflow execution traces. In: Groth, P., Frew, J. (eds.) IPAW 2012. LNCS, vol. 7525, pp. 82–96. Springer, Heidelberg (2012)

[CAA07] Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. In: Arenas, M. (ed.) DBPL 2007. LNCS, vol. 4797, pp. 138–152. Springer, Heidelberg (2007)

[CBHS05] Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: The 14th International Conference on World Wide Web, New York, NY, USA, pp. 613–622 (2005)

[CCPR13] Cheah, Y.-W., Canon, R., Plale, B., Ramakrishnan, L.: Milieu: lightweight and configurable big data provenance for science. In: 2013 IEEE International Congress on Big Data (BigData Congress), pp. 46–53, June 2013

[CCT09] Cheney, J., Chiticariu, L., Tan, W.-C.: Provenance in databases: why, how, and where. Found. Trends Databases **1**, 379–474 (2009)

[CP12] Cheah, Y.W., Plale, B.: Provenance analysis: towards quality provenance. In: 8th IEEE International Conference on eScience, October 2012

[GP13] Ghoshal, D., Plale, B.: Provenance from log files: a bigdata problem. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, New York, NY, USA, pp. 290–297 (2013)

[GT12] Gehani, A., Tariq, D.: SPADE: support for provenance auditing in distributed environments. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 101–120. Springer, Heidelberg (2012)

[HZ09] Hartig, O., Zhao, J.: Using web data provenance for quality assessment. In: The Workshop on Semantic Web and Provenance Management at ISWC (2009)

[Mea05] Wong, S.C., Miles, S., Fang, W., Groth, P.T., Moreau, L.: Provenance-based validation of e-science experiments. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 801–815. Springer, Heidelberg (2005)

[MRHBS06] Muniswamy-Reddy, K.-K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware storage systems. In: The Annual Conference on USENIX 2006 Annual Technical Conference, Berkeley, CA, USA (2006)

[NOZ+06] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Memik, G., Choudhary, A.: Minebench: a benchmark suite for data mining workloads. In: 2006 IEEE International Symposium on Workload Characterization, pp. 182–188, October 2006

[SP11] Simmhan, Y., Plale, B.: Using provenance for personalized quality ranking of scientific datasets. Int. J. Comput. Appl. (IJCA) **18**(3), 180–195 (2011)

[Yea12] Yang, H., Michaelides, D.T., Charlton, C., Browne, W.J., Moreau, L.: DEEP: a provenance-aware executable document system. In: Groth, P., Frew, J. (eds.) IPAW 2012. LNCS, vol. 7525, pp. 24–38. Springer, Heidelberg (2012)