Technical Report

UCAM-CL-TR-930
ISSN 1476-2986

Number 930

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Provenance-based computing

Lucian Carata

December 2018

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2016 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

ISSN 1476-2986

Provenance-based computing
Lucian Carata

Summary

Relying on computing systems that become increasingly complex is difficult: with many factors potentially affecting the result of a computation or its properties, understanding where problems appear and fixing them is a challenging proposition. Typically, the process of finding solutions is driven by trial and error or by experience-based insights.

In this dissertation, I examine the idea of using provenance metadata (the set of elements that have contributed to the existence of a piece of data, together with their relationships) instead. I show that considering provenance a primitive of computation enables the exploration of system behaviour, targeting both retrospective analysis (root cause analysis, performance tuning) and hypothetical scenarios (what-if questions). In this context, provenance can be used as part of feedback loops, with a double purpose: building software that is able to adapt for meeting certain quality and performance targets (semi-automated tuning) and enabling human operators to exert high-level runtime control with limited previous knowledge of a system's internal architecture.

My contributions towards this goal are threefold: providing low-level mechanisms for meaningful provenance collection considering OS-level resource multiplexing, proving that such provenance data can be used in inferences about application behaviour and generalising this to a set of primitives necessary for fine-grained provenance disclosure in a wider context.

To derive such primitives in a bottom-up manner, I first present Resourceful, a framework that enables capturing OS-level measurements in the context of application activities. It is the contextualisation that allows tying the measurements to provenance in a meaningful way, and I look at a number of use-cases in understanding application performance. This also provides a good setup for evaluating the impact and overheads of fine-grained provenance collection.

I then show that the collected data enables new ways of understanding performance variation by attributing it to specific components within a system. The resulting set of tools, Soroban, gives developers and operation engineers a principled way of examining the impact of various configuration, OS and virtualization parameters on application behaviour.

Finally, I consider how this supports the idea that provenance should be disclosed at application level and discuss why such disclosure is necessary for enabling the use of collected metadata efficiently and at a granularity which is meaningful in relation to application semantics.

Personal publication list

Significant contributions

[Lucian-1] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper. “A Primer on Provenance”. In: *Commun. ACM* 57.5 (May 2014), pp. 52–60 (cit. on p. 19).

[Lucian-2] Lucian Carata, Ripduman Sohan, Andrew Rice, and Andy Hopper. “IPAPI: Designing an Improved Provenance API”. In: *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*. TaPP ’13. Berkeley, CA, USA, Jan. 2013, 10:1–10:4 (cit. on p. 111).

[Lucian-3] James Snee, Lucian Carata, Oliver R. A. Chick, Ripduman Sohan, Ramsey M. Faragher, Andrew Rice, and Andy Hopper. “Soroban: Attributing Latency in Virtualized Environments”. In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, July 2015 (cit. on pp. 79, 94).

Secondary contributions

[Lucian-S4] Sherif Akoush, Lucian Carata, Ripduman Sohan, and Andy Hopper. “MrLazy: Lazy Runtime Label Propagation for MapReduce”. In: *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. Philadelphia, PA, June 2014.

[Lucian-S5] Oliver RA Chick, Lucian Carata, James Snee, Nikilesh Balakrishnan, and Ripduman Sohan. “Shadow Kernels: A General Mechanism For Kernel Specialization in Existing Operating Systems”. In: *6th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 15)*. ACM, 2015 (cit. on p. 48).

Technical reports

[Lucian-TR6] Lucian Carata, Oliver Chick, James Snee, Ripduman Sohan, Andrew Rice, and Andy Hopper. *Resourceful: fine-grained resource accounting for explaining service variability*. Tech. rep. UCAM-CL-TR-859. University of Cambridge, Computer Laboratory, Sept. 2014.

Acknowledgements

I would have certainly not finished this thesis if it weren't for the support (both moral and practical) of the people who have been around me those four years. I am grateful for their help and honored to have met them. Good advice is hard to come by, and I sometimes feel like I received more than my fair share – but then again, such is the curse of spending your time amongst people smarter than you.

I cannot start without thanking my supervisor, *Andy Hopper*, who has encouraged me from day one. Throughout my PhD, I had the feeling of him being the invisible hand steering me away from dead-ends while pointing in the right direction; not strong enough for me to feel I'm without free choice or research thought, but there to make me properly justify my line of enquiry (and realise my foolish ways). Nevertheless, I always emerged from our meetings feeling better about both myself and the research I'm doing. Without his constant advice and financial support I'm certain this thesis would have remained a dream of mine.

I have received considerable help and support from *Ripduman Sohan*, during our almost daily interractions. I must sincerely thank him for putting up with me! His extensive knowledge of the Linux kernel and of system measurement in general have proven irreplaceble in shaping the results in Chapter 3. More generally, our discussions have clarified for me what problems are important and worth working on, and have given me the courage of exploring areas which I hadn't considered to be my strong points. The style of debating and generating solutions through numerous group brainstorm sessions proposed by Rip have refined my ideas around too many topics to count. Furthermore, his direct feedback regarding possible improvements to the thesis has led to a significantly better end-result.

Thanks are due to *Andrew Rice*, who has helped me throughout my first year here and forced me to read lots of research papers so that I can avoid working on problems that have already been solved. *Robert Harle* and *Alan Mycroft* were the first to read and evaluate my research plans; their suggestions and doubts have shaped the argument for computational provenance as is currently presented. *Chris Town* has given me an outside perspective on how my work could be applied in other areas of computer science, and discussing the topic with him made my own ideas more focused and clear. The numerous at-whiteboard discussions I had with *Sherif Akoush* regarding designing provenance-enabled software, have certainly influenced the implementations of the systems presented in this thesis, and I thank him for his openness and insights.

To my examiners, *Robert Watson* and *Paul Watson*, who have provided constructive feedback and were genuinely interested in the topics I have presented. The final version of this thesis is significantly improved due to their advice.

To my Resourceful partners-in-crime, *Ollie* and *James*: I felt that contributing to a common project has definetely made our overall results better, and has allowed us to support each other when things looked as if they would never work. I'll surely remember the nights spent eating takeaway food and working until dawn together with you guys, it has definitely motivated me to stay focused on getting things done. Having a debate with Ollie certainly challenges all your opinions about what is novel, well designed and whether it's worth doing (as well as what is right, wrong and meaninful in life). As for James, I'm sure he'll show the guys at Apple that our approach is The Right Way$^{\text{TM}}$.

To my office colleagues, Tom and Nikilesh (as part of the Fresco team): I must thank you for the discussions on possible research and ideas for the future of provenance. Our brainstorms have certainly helped in shaping how I think about provenance and its role in computing moving forward. Now, we must let everybody else know as well...

I would also like to thank EPSRC, The Cambridge Computer Laboratory and the Cambridge Home Scholarship Scheme (CHESS) for funding my research.

On a more personal note, I have certainly benefited from the moral support and understanding of friends and family during my time as a PhD student. I would like to thank *Mariana*, without whom I would have never applied to do a PhD in Cambridge in the first place. *Bogdan Roman* has always been there for me with good advice on research and life in general; furthermore, he was offered me plenty of opportunities to contribute to side-projects whenever work on my PhD seemed too daunting.

To *Cristiana*, I'm glad to have found you; I thank you for being supportive during the long-winded process of me writing up: sometimes, I felt you understood me better than I understood myself - and I can only admit that when all else failed, you have kept me motivated with your positive energy and optimism.

Last but not least, I want to thank my Mom and Dad, who have always been there for me with a good word of encouragement - I can only hope they know how much I appreciate them and value their advice; they have set the example and made me want to be a wiser person. It remains to be seen if I shall ever succeed in *that* endeavour.

Contents

“In the beginning there was nothing, which exploded.”
— Terry Pratchett, Lords and Ladies

1 Introduction

WHEN ASSESSING THE QUALITY or validity of a piece of data, humans are not considering it in isolation. Instead, we instinctively examine the context in which it appears, try to determine its original sources or review the process through which it was created. However, going over those stages for *digital data* is not straightforward: the results of a computation might have been derived from numerous sources and by applying complex successive transformations, possibly over long periods of time and across different computers.

As the quantity of data that contributes to a particular result increases, it becomes harder to keep track of how different sources and transformations are related to the final result. Without automated mechanisms for making sense of the *history* of such results, we are constrained in our ability to answer important questions about them, such as: what were the underlying assumptions on which the result is based? under what conditions does it remain valid? why does / doesn’t the data contain particular values? what other results were derived from the same data sources?

The metadata that needs to be systematically captured in order to answer those (or similar) questions is called *provenance*[1] and refers to a graph describing the relationships between all the elements (sources, processing steps, contextual information and dependencies) that contributed to the existence or to the properties of a piece of data.

A number of use cases are representative for the practical use of provenance data. The current thesis focuses on using it to explain observed behaviour across the software stack (explaining performance degradation, errors and attributing the usage of resources to different components in a complex system). However, in order to understand why a provenance-based approach offers new insights towards solving such issues, it is useful first to consider a wider context:

Where does data come from?

Consider the need to understand the conditions, parameters or assumptions behind a given result; in other words, being able to point at a piece of data (research result, anomaly in a system trace or a value in a published report) and ask: where did it come from? Answering this question would be useful for any experiments involving digital data, such as “in silico” experiments in biology, other types of numerical simulation or system evaluations in computer science.

[1]or lineage; we will consider the two terms equivalent in this thesis. Some authors have historically made a distinction between provenance (referring only to original sources) and lineage (referring to the derivation history for a data product) [21]

The provenance for each run of such experiments contains the links between results and corresponding starting conditions or configuration parameters. This becomes important especially when considering future usage of data, with early results being used as the basis of further experiments. Manually tracking all the parameters from a final result through intermediary stages and to original sources is burdensome and error-prone. When considering systems research and the goals of this thesis, an added difficulty is the interaction *between* various components that contribute to an observed value (e.g. latency), and the nature of such contributions: could provenance be used in identifying the primary tunable parameters that control the system's behaviour?

Of course, systems researchers are not the only ones requiring this type of tracking. The same techniques could be used to help people in the business or financial sectors, for example in figuring out the set of assumptions behind the statistics reported to a board of directors, or in determining what mortgages were part of a traded security.

Knowing where a piece of data comes from can also be helpful in complex decision-supporting scenarios: If a (semi-)automated algorithm provides answers or computes results which are then used to take decisions that could affect humans, provenance offers ways of both explaining what the algorithm did, understanding why errors might have occurred and it could provide the mechanisms for correcting or isolating those errors.

To find concrete examples, we should look at algorithms used to compute credit scores or insurance costs by integrating data from numerous sources, or at adaptive machine learning algorithms controlling self-driving cars. Normal humans interacting with such algorithms will have a very superficial understanding of their internal operation (much less so than a researcher running an experiment). Therefore, making those algorithms record provenance will add significant value in being able to explain their outputs. Establishing feedback loops by feeding the provenance of outputs as an input to further processing means that the algorithms themselves can adapt and automatically try to isolate errors. Alternatively, the provenance feedback loop channel could be used to retain a form of human control over such systems while not compromising their automated nature.

Who is using this data?

Instead of tracking a result back to its sources, we can capture provenance to understand where that result has been subsequently used or to find out what data was further derived from it. Some existing domain-specific mechanisms already implement very coarse-grained variants of such tracking: for example, the recording of citations for a given article or various web pingback mechanisms.

However, provenance would significantly increase the usefulness of such data by making similar queries available at a much finer granularity.

A company might want to identify all the internal uses of a certain piece of code in order to respect licensing agreements or for keeping track of code still using deprecated/unsafe functions that need to be removed. The general security scenario of having to identify non-obvious attempts of leaking sensitive data is also within scope: Take an attacker who tries to hide in the noise and not trigger network security monitoring solutions while exfiltrating data. A possible strategy would be splitting a file into n small pieces and sending them from multiple servers in a network to various cloud VMs under the attacker's control, over a long period of time (months or years). Here, provenance could help by identifying where data from the sensitive file ended up (in the n pieces sent over the network) and what techniques were used by the attacker in doing so.

Using similar mechanisms, end users should be able to track what personal information is used by a mobile application and determine whether it is only displayed locally or sent over the network to a third party. However, this also raises data access enforcement questions which will need to be clarified: if systems are designed to allow individuals to strongly enforce constraints on what and how personal information is used by third parties using provenance, then the same primitives will enable the implementation of strong DRM schemes (the third parties could do the same with the data they make accessible to the user). In turn, those might limit the ability of users to use legally purchased digital data across different devices (for example). However, this symmetry does not necessarily exist if what end users require is just a proof that data was used for allowed purposes, without gaining access to its full provenance.

The same use case covers the general propagation of erroneous results, when we need to understand what pieces of data have been invalidated by the discovery of an error and what re-computations need to take place for updating derived results.

How was it obtained?

Provenance can also be used to get a better understanding of the *actual process* through which different pieces of input data are transformed into outputs. This is important in situations where computer engineers or system administrators need to debug the problems arising when running complex software stacks.

Understanding such issues becomes feasible because, when it is possible to differentiate between a correct and an erroneous system output, comparing their provenance will point to a list of potential root causes of the error. In more complex scenarios, the issues might not be directly linked to particular outputs but to an (undesired) change in behaviour. Tracking system intrusions or explaining why the response tail latency has increased by 20% for a server are good examples. In those cases, grouping outputs with similar provenance could be used for identifying normal versus abnormal system behaviour and for explaining the differences between the two. Such problems are the focus of the current thesis, and I refer to the type of provenance collected to solve them as *computational provenance* (provenance referring to properties of computations such as performance or the appearance of errors).

Another important application of being able to examine the process through which particular results were obtained is in enforcing policies that those outputs should follow. Taking the example of some Volkswagen cars having algorithms that detect test conditions and adjusting system parameters as to fall into admissible emission regimes: one can envision tools that examine the provenance graph of the test result, which will include the provenance of any car-specific parameters during the test, to determine whether the engine configuration is significantly different from configurations normally used when driving the car. This would of course require some form of manufacturer transparency in regards to the algorithms used to decide upon various system parameters. However, such forms of transparency can be imposed by regulations and standards compliance requirements.

Solving the problems highlighted by those use-cases is required if we aim for the ability to deal with even more complex situations. Those remain beyond the scope of the work described in this thesis, but are important in terms of the bigger picture: as sensors become pervasive and machine learning algorithms get involved in optimising our digital infrastructure, non-experts will still want to feel confident and in control of the underlying computing systems.

The management of smart homes and self-driving cars are two examples of this. What “knobs” will be available to human operators for providing corrective inputs? For example, in the case of driverless cars, a human could be asked to provide corrective inputs like, “too aggressive” or “wrong lane switch”. However, understanding why such things happened in the first place will require some form of provenance.

Provenance systems

Together, the three use cases provide an overview of the ideal provenance application space, but do not describe the technical details involved in making those applications possible. To realise each scenario in practice, one or more *provenance systems* need to be integrated into the data processing workflow, becoming responsible for capturing provenance, propagating it between related components and making it accessible to user queries or automated analysis tools.

In many ways, people might already be running a very specialised version of such a system today: all auditing, tracing frameworks or change tracking solutions will collect some form of provenance, even though they might not identify it as such. The advantage of thinking about provenance as a standalone concept is the ability to use this metadata in a principled way, allowing result verifiability and complex historic queries irrespective of the underlying mechanisms used to collect it and across applications or software stacks. This implies that provenance data would become something you can *perform computations on*.

Research Thesis

A complex heterogeneous system that is provenance aware can detect and take automated steps to recover from situations in which it produces results deviating from a standard (of performance, correctness, security), using provenance metadata to evaluate and understand its current state.

In supporting the thesis I will design, implement and evaluate a framework allowing the capture of provenance-enhanced system measurements, showing how those can be used in analysis for understanding application performance properties.

Outlook and contributions

The goal is to make a strong case for considering provenance a fundamental primitive of computation, that enables the exploration of system behaviour targeting both retrospective analysis (root cause analysis, performance tuning) and hypothetical scenarios (asking what-if questions). In this context, provenance can be used as part of feedback loops, with a double purpose: building software that is able to adapt for meeting certain quality and performance targets (semi-automated tuning) and enabling human operators to exert high-level runtime control with limited previous knowledge of a system's internal architecture.

My contributions towards the stated goal are threefold: (i) providing low-level mechanisms for provenance collection taking into account OS-level resource multiplexing, (ii) proving that this provenance data can be used in inferences about application behaviour by attributing resource consumption to relevant software-level activities and (iii) showing how the same data, together with a causal model describing assumed relationships between system components, can be used in determining metrics for hypothetical scenarios ("what-if" questions).

In order to build the primitives required for such use cases in a bottom-up manner, I first describe the traditional provenance landscape (Chapter 2), while also introducing new terms and concepts (computational provenance, the n-by-m problem). The systematic discussion of issues surrounding provenance systems is new and considers topics in capture (data granularity, layering, versioning), querying (classifying types of query languages for provenance) as well as overheads and security models. The second part of the background chapter ties the provenance world with current systems research in tracing, instrumentation (as low-level mechanisms), profiling, performance measurement (as specialised applications) and root cause analysis in the context of complex/virtualized systems (as analysis targets). These are relevant in contrasting non-provenance based ideas to the tools and methodologies developed in the following chapters.

The key lacking aspect is identified as the difficulty of understanding how global system measurements are related to activities performed by running applications. Answering this problem, in Chapter 3 I describe Resourceful, a framework that enables capturing OS-level measurements in the context of application activities. It is the contextualisation that allows tying the measurements as part of provenance in a meaningful way, and I look at a number of use-cases in understanding application performance. This provides a good setup for evaluating the impact and overheads of fine-grained provenance collection.

I then show that the collected data enables new ways of understanding performance variation by attributing it to specific components within a system (both on bare metal and under virtualization). The resulting set of tools, Soroban (Chapter 4), gives developers and operation engineers a principled way of examining the impact of various configuration, OS and virtualization parameters on application behaviour. An approach is described for undertaking performance evaluation, considering the causality relationships that exist between different captured metrics, as estimated by a domain expert. The end result can give answers to questions such as: what would the latency of this HTTP request have been, had it been serviced by a server running on bare metal instead of inside a virtualized environment (with contention from other VMs)?

The thesis concludes with an in-depth look at future directions for provenance in systems research (Chapter 5), relating to improvements that can be made to Resourceful and Soroban, as well as discussing generalized provenance APIs and tools for exploring computational provenance results.

"Straight ahead of him, nobody can go very far . . . "
— Antoine de Saint-Exupéry, The Little Prince

2 Background[1]

HISTORICALLY, provenance systems were the focus of research in the database field, with the aim of understanding how and when materialized views should be updated in response to changes in the underlying tables [41]. Because of the well defined relational model, it has proven possible to both derive precise provenance information from queries [25] and to develop formalisms which allow its concise representation [64]. This has been further extended in systems such as Trio [146], allowing records to have an associated uncertainty and being able to propagate it across multiple queries by using provenance.

In contrast, capturing provenance for applications performing arbitrary computations (with possible side effects and not restricted to a particular set of valid transformations) has proven more challenging. Research efforts in this area have focused on the collection of provenance at particular points in the software stack (by modifying applications, the runtime environment or the kernel).

Figure 2.1 presents a general timeline of systems and the current chapter discusses the characteristics of some that are representative for a larger class of solutions in the design space.

OS level: PASS [100, 102], SPADE [61] and CAMFLOW [111] investigate capturing provenance by observing application events such as process creation or IO. Those are then used for inferring dependencies between different pieces of data. Subsequent versions of those systems have also added the ability to integrate with special user space libraries in order to obtain more information about application behaviour. While low-level, those tools are concerned with tracking the provenance of data as it is being processed (the focus is on the history of the data). Unlike them, the work presented in this thesis deals with the provenance of the computations and transformations applied to the data (as operations that in themselves have a history and interesting properties). I define the latter as *computational provenance*, and in particular will be interested in making use of the provenance of system measurements as a way of characterising computation behaviour (performance variation, bottleneck shifting, errors, etc). The design, implementation and evaluation of such a system, called *Resourceful*, is discussed in detail in the following chapters.

Workflows: Vistrails [125] and ZOOM [15] are workflow management systems with the ability to track provenance for the execution of various workflows and (in case of Vistrails) for the evolution of the workflows themselves. Such systems are only able to capture *complete* provenance

[1]An earlier version of this chapter was published as part of the CACM and ACM Queue articles [Lucian-1]

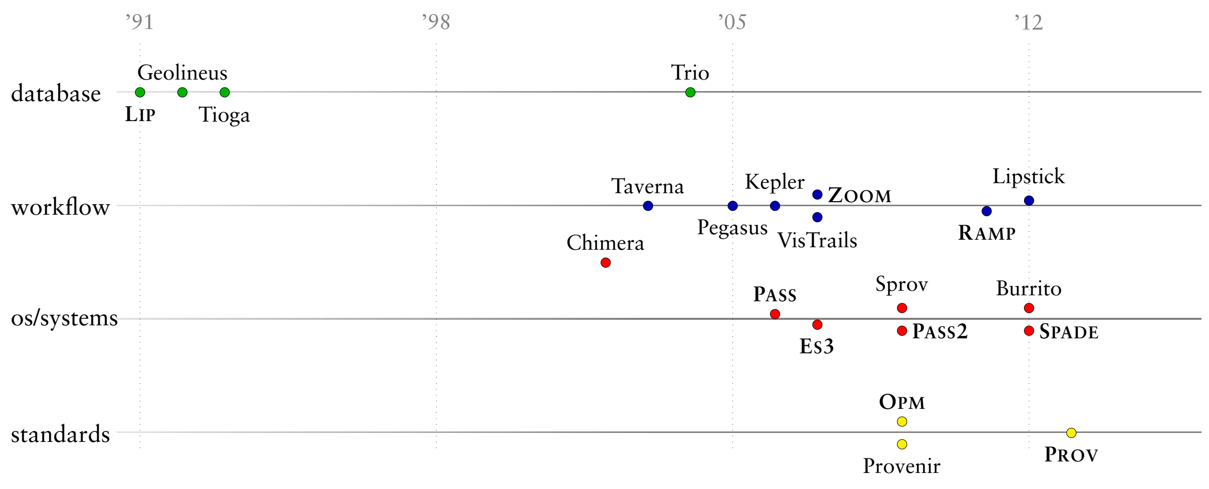


Figure 2.1: A timeline of provenance systems. *Databases* have been the traditional space for provenance research. Scientific communities using *workflow* applications have been the first to drive the adoption of specialized provenance systems. The OS/*systems* community has been pushing for more general systems that would ultimately work irrespective of the computational environment. Simultaneously, the semantic-web community has been working on *standardizing* provenance on the web.

if all tasks are defined and executed within or under the control of the workflow engine.

Application level: Burrito [68] tracks user space events, while also supporting additional user-provided annotations. SPROV [72] focuses on the security of provenance, and provides a thin wrapper around the standard C IO library. A newer version is capable of using provenance captured by other systems, such as PASS. Similarly, SPADE is able to accept domain-specific provenance from arbitrary sources through reporter plugins. A recurring theme of this thesis, capturing the *context* of an application at the time the OS performs actions on its behalf, turns out to be essential in accurately attributing resource consumption and making complex inferences about application performance. The aim for making provenance generally useful should be one of integrating application-level data with OS-level data, in ways which allow each to be understood in terms of the other (for example, understanding the kernel-level actions triggered by an application executing a SQL query or writing a file on an NFS partition).

Big data: Lipstick [5] and RAMP [110] both tackle the problem of tracking provenance in big data scenarios (Map-Reduce jobs).

It is the properties of those systems that are important for understanding what can be recorded and with what trade-offs, overheads and security implications. The design choices, together with outstanding issues, are relevant in the context of computational provenance (the focus of this thesis) and inform the design of the systems and diagnosis methodologies developed in Chapters 3 and 4.

2.1 Provenance system properties

From the perspective of adding provenance-recording abilities to applications, there are a number of aspects that are important when looking at a provenance system:

- **What can it capture?** Understanding what metadata is relevant for a series of data transformations and how the captured information enables and limits the type of questions that could be asked later. The properties of the system doing capture have themselves a strong influence on how the data can be used: for example, the lack of robustness on failure (or even simple event dropping) will make the collected provenance unusable for security purposes.

- **Integration effort:** Integrating the system within existing data processing scenarios might involve the need to run on a special OS kernel, make changes to the runtime environment or link applications with provenance libraries.
- **How to answer queries using provenance?** The way one might explore and ask questions based on the captured metadata is important for understanding how provenance can be used to satisfy the various use cases. When capturing provenance data for a well-defined purpose, it is useful to think of provenance queries in relation to traditional audit reduction tools: out of all recorded elements, only a few might be of interest, and both query time and storage savings can be obtained by reducing the data while having in mind its use.
- **Understanding overheads**: Given one use case, it is often essential to grasp whether the overheads imposed by running the provenance system are acceptable, and to be able to predict how those overheads will scale as a function of the number of data dependencies and transformation steps executed. In order to reduce overheads, capture systems could be configured to perform various forms of aggregation or be subject to retention policies, either directly at the time of capture or asynchronously (similar to audit systems [83])
- **Security issues:** provenance metadata will often require different access controls from those of the data itself, and it is important to understand the security concerns raised by the use of a particular provenance system.

Based on the above features, I can categorise the properties of the systems picked as representative, referring back to the motivating use cases as needed. For an orthogonal view, a per-system summary of properties can be consulted in Table 1.

2.1.1 What can it capture?

The metadata captured by provenance systems typically relates the state of digital entities (files, tables, programs, network connections etc.) at different stages in their lifetime to historic dependencies on other entities or processes. For computational provenance, this means the identification of links between a series of actions (for example an administrator command triggering a backup job) and their side-effects on particular applications (i.e. performance of end-user requests). In both cases, two concepts are fundamental for determining what is captured and how: *granularity* and *layering*.

Granularity

The *granularity* of capture refers to the size of basic primitives that accumulate provenance within a system. Consider a scientist who uses a configuration file storing various experiment parameters as one of the inputs to a simulation program. Capturing provenance at file granularity will discover the dependency between the simulation program and the configuration filename. However, the scientist is interested in understanding the relationships between the simulation results and individual parameters in the file, which requires capture at sub-file granularity.

The exact meaning of varying granularity from fine to coarse also depends on the underlying data model of the application. For example, database provenance systems could store provenance metadata for an entire table, a row within the table, or for each cell. Provenance capture at the table level is coarse grained and can answer questions such as "From which other tables has table

X derived its data?". Finer granularities would determine the relationships between individual rows or cells. Of course, multiple granularities can be considered at the same time.

Systems such as PASS [23], capture provenance by intercepting system calls made by applications as they execute. At this level, provenance is fine grained and can provide a detailed image of an application's execution and dependencies.

However, the *noise levels* in the collected data are also elevated, making it harder to extract useful information: Consider a python script that copies one file to another. When running the script, the python interpreter will first read and load any required modules from disk. Thus, beyond the dependency on the actual input, the final provenance graph will link the output file to all the python modules used by the interpreter. This extra data can make it difficult to sift through the provenance graph as an end user, so generally heuristics are needed to determine which entities are important and which should be ignored.

Workflow systems such as Vistrails [125] avoid the noise problem and can capture provenance at any granularity because the processing steps and their dependencies are explicitly declared by the end user. However, such systems are also inherently limited to *only* recording data transformations that were part of the defined workflow.

The n-by-m problem Independent of the system that is chosen, it may not be possible to accurately determine the dependencies between input and output data. This is illustrated by the *n by m problem*, where a program reads N input files and writes M output files. Even when tracing system calls for individual reads and writes, it's not possible to infer which reads affected a particular write, so the provenance graph has to link each output file to all of the inputs. A system that is unaware of the semantics of individual data transformations within a process will always present a number of such false positive relationships. Both PASS and Vistrails have this problem, as they treat the process or each workflow step as a black box.

The *n by m problem* can be solved by capturing provenance at even finer granularities. This can be done using binary instrumentation techniques [124] and computing the provenance of the output as a function of the executed code path and data dependencies. Even if this method requires no modification of the application, the tradeoff is a significant increase in space and time overheads. A low-overhead alternative would be to modify the application to explicitly disclose relevant provenance using an API such as CPL [89], but this requires additional effort from the developer, as further discussed in section 2.1.2.

Granularity is not the only aspect a user needs to think about when determining his/her requirements for a provenance system. It is just as important to know in which layer the provenance collection takes place.

Layering

Provenance metadata can be captured at multiple layers in the stack i.e. for the application, middleware (runtime/libraries), operating system and/or in hardware. Capturing provenance across multiple layers provides users with the ability to reason about their data and processes at different *levels of abstraction*, with each layer providing a different view on the same set of events happening in the system.

For example, consider copying rows between two tables in a spreadsheet and saving the result. A system collecting provenance at the OS layer will observe a number of IO operations to/from the file. However, the notions of "tables" and "rows" are only known to the application, and dependencies amongst them cannot be inferred from the metadata collected by lower layers. If querying for such relationships is needed, provenance must be captured in the application layer as well.

Cooperation between layers When requiring provenance capture at multiple layers, a practitioner could choose a different (specialised) provenance system for each layer in the stack or a single provenance system that was designed to span capture across multiple layers.

In both cases, multiple provenance-aware components must cooperate by communicating different pieces of metadata between layers. This can be achieved either by adhering to a common provenance data model (such as OPM [99] or PROV-DM [98]) or by providing an universal API and allowing each component to both accept and generate provenance using it. PASSv2 provides a a disclosed provenance API (DPAPI) that can be used for this purpose.

However, a second issue exists. Merely collecting metadata at different layers will result in islands of provenance, unrelated to each other. For an actual *mapping* of provenance objects between layers, all entities describing the same application-level activity/event (i.e user copying some text) must be grouped, for example by tagging them with an unique identifier.

SPADEv2 for instance uses a multi-source fusion filter (with process id as a tag) to combine provenance data from multiple sources describing the same event at the same level of abstraction. When provenance is reported at different levels of abstraction SPADEv2 uses a cross-layer composition filter that explicitly connects a set of lower-level operations to a higher-level concept through an *isAbstractedBy* edge.

The issue of mapping between layers is crucial for computational provenance, as diagnosing performance variations or application errors relies on inferring the components which are to blame, and those might exist at different levels in the software stack.

Data versioning Provenance collection in a given layer typically involves capturing the chain of events performed by the application on a given piece of data. However, this does not necessarily require the system to capture multiple versions of data as it is being transformed. Assume that a user edits a file using a text editor on a PASS enabled system. The provenance metadata saved by PASS can provide information such as the program used to edit, number of bytes written to the file etc. But it is not possible to revert the file back to a previous state or know what the actual data changes were. In cases where the current contents of the file depend on values in previous versions, provenance systems need to store versions of data besides processing events in order to assure verifiability. Because of this, provenance systems such as Burrito [68] not only track system call level events, but are also running on top of a versioning file system. Other systems such as Lipstick and RAMP do not require versioning as they run on top of append-only file systems (all versions are implicitly stored)

Versioning can prove expensive when done for certain layers in the stack. For example, deciding to version data in the hardware layer (versioning the values or a register) would create large amounts of data, and should be preceded by an evaluation of actual benefits. In other cases, versioning can actually improve the collection of provenance. This is the case in application layer, where a user can undo/redo actions. Most GUI applications provide this functionality by default and intercepting the undo stack has been shown to be viable method for automatically inferring provenance [27].

2.1.2 Integrating provenance into existing workflows

The effort needed for integrating a new piece of technology within an existing workflow is an important practical criteria when choosing a provenance system. This measures how much the provenance system will intrude on user's normal working practices, and a cost-benefit analysis should be made depending on the use case.

Some systems impose larger upfront expenses due to how they collect metadata. For example, they ask the application to explicitly attest to provenance information, as is the case with APIs that allow you to supply annotations about the actions being executed. An example of this is the PASSv2 DPAPI, which offers augmented read and write calls to which one can pass data indicating the meaning of the read or write call that is being made. The end result is an increase in the development effort, as code must be updated to call the new API. All future code changes must also keep the provenance-related code in sync, and failing to do so will most likely cause invalid metadata to be captured.

Similarly, systems such as ZOOM or Vistrails ask you to declare the entire workflow in advance and can only track dependencies that run on top of their execution engines. Subsequent work must be done within the same system if dependency links need to be maintained.

As a group, the literature refers to those as *disclosed provenance* systems, and they are recognised for their ability to offer improved semantic descriptions of provenance. However, the trustworthiness of the provenance captured in this way is a concern when running in untrusted environments. The reasons for this are twofold: because of how disclosed provenance works, there are no explicit guarantees that what is declared is actually what is happening (code declaring provenance can get out of sync with actual program code). More subtly, compromised applications can be made to perform different operations after they have called a function disclosing provenance: as long as an attacker is aware that provenance is being recorded, he can prevent the process from exposing his actions. This means that both the original developers and the running application need to be trusted if the resulting provenance is to be relied on. A detailed discussion on the ways an attacker could circumvent the provenance capture system is done by Balakrishnan [7].

In terms of computational provenance, the highest risk is presented by shared environments (such as VMs collocated on the same physical machine). There, gathering detailed measurements about kernel and hypervisor activities could reduce the privacy of workloads running concurrently (mechanisms such as workload fingerprinting could be employed to detect and classify neighbour workloads). As with other types of provenance, this means that the resulting measurements should have separate authorization controls when compared to data normally produced by applications. This has the potential of adding yet another burden in terms of integration and need of institution/company policies.

Other provenance systems aim to reduce the overhead imposed on the user. These tend to take a different approach by observing the users' applications, recording information about how these applications interact with each other and the rest of the OS and inferring provenance based on it. They are often referred to as *observed provenance* systems. Systems such as PASS that intercept system calls made by a program or others such as SPADE that can hook into the audit sub-system in the Linux kernel to observe the program's actions are examples of this type of system. They tend to have the lowest intrusiveness. Often once the system is installed a user can proceed as usual while having provenance captured for all of the operations they perform. However, observed provenance systems have their own shortcomings, mostly due to the loss of semantic information when treating each process as a black box.

2.1.3 Answering questions based on provenance

Using a provenance system is only as useful as the questions that one can answer based on the collected metadata. However, querying is recognised as a challenging problem: users often want to query over a broad range of information or they ask questions that the designers of a provenance system did not anticipate; depending on the granularity of capture, there might be either insufficient data to respond to a query, or the system might produce so much data that it is difficult explore and understand it. From the research performed to date in the field, two core paradigms of querying have emerged and a smaller number of systems use some hybrid of both approaches.

Exploratory

The first major paradigm is exploratory query, which takes advantage of the human ability to spot patterns. This is important when users don't have an exact idea of what metadata they might want to retrieve. Exploratory systems are usually characterised by presenting the user with a visual representation of the provenance graph and giving them tools to better explore it without succumbing to information overload. This is a notably hard problem given that even small provenance graphs can easily contain thousands of nodes. A number of the approaches taken involve either exploring subgraphs based on contextual filtering (such as InProv [20]) or intelligent clustering methods. An example of the latter is the PASS Map Orbiter [90] viewer, which implements an algorithm for dynamically summarising nodes allowing you to expand and contract areas of detail while browsing.

Directed

The second major paradigm is directed query, an approach more closely linked to the classic field of database query. It requires the user to express questions about the provenance of data as queries in a language that is often a specialised extension of SQL or of a path query language.

The approach is effective if the users know precisely what information they require, but unlike exploratory methods it does not facilitate discovery of new insights about the provenance graph.

vtPQL [125] is an example of the directed approach used in the Vistrails system. The language is designed to enable the user to express provenance queries about three different aspects of the workflow: the execution log, the abstract workflow representation and the evolution of the workflow in time. The querying system allows a user to specify restrictions on all three of these spaces simultaneously. For example restricting the execution logs to a particular day, highlighting a single workflow module and choosing a particular version of the workflow. This is helpful as it allows the user to think in terms of orthogonal querying concerns.

Hybrid

Some systems use a hybrid of the two paradigms. For example the ZOOM system [15] starts from a user-provided 'declaration of interest' to derive a contextually appropriate minimal from of the provenance graph. The heart of the system is an algorithm that summarises 'irrelevant' parts of the graph in ways that maintains their semantics. The user only needs to provide the list of the modules in the workflow definition that are of interest, and is then allowed to browse the provenance graph without being distracted by unimportant pieces of information.

2.1.4 Understanding overheads

As with any computational functionality, provenance capture has associated temporal and spatial costs. Given that provenance support is likely to be an *additional* consideration to the primary function of the system, only leveraged when the lineage properties of the data are required, it is imperative to minimise the overheads.

General purpose provenance systems typically capture either (disclosed) evolutions of a given workflow or (observed) low-level operations carried out by executing processes. Broadly speaking, the time and space overheads for capturing the provenance of workflow evolution is proportional to both the number of changes in the workflow *and* the number of times a workflow is executed. In comparison, the provenance overhead of capturing an execution log is proportional to the number of recordable operations executed.

Time overheads In practice the provenance capture cost of workflow systems (and by extension of other disclosed provenance systems) is minuscule due to their limited approach to collecting running process information. Both Zoom and Vistrails, for example, report an approximately 1% increase in execution time [15, 57].

For systems that record process execution, provenance capture costs are a function of the cost of intercepting and recording observable operations. While intuitively it may appear that provenance capture at the operation level is prohibitively expensive from a temporal perspective, reported results show that this is not the case. Kernel based system call interception mechanisms such as in PASSv2 have a 1–23% overhead on workloads representative of real-world applications [100, 102]. Similarly SPADEv2, which utilises kernel auditing infrastructure for provenance capture, reports $< 10\%$ overhead on Windows, Linux and OS X for production Apache runs [61].

For I/O heavy workloads, however, provenance capture may impose larger overheads. PASSv2 for example, reports up to 230% overhead on small file benchmarks [102], but the absolute increase in execution times remains small.

The interception mechanism can also significantly influence provenance capture overhead in this regard. SPADEv2 for example, supports operation interception via the kernel auditing mechanisms on OS X while on Windows it requires a file system filter driver that relays operations to the provenance collector. As a consequence, provenance enabled Apache builds are 50% slower on Windows but only 5% slower on OS X.

The temporal cost of recording operations may also be of potential concern where provenance is being recorded at an extremely fine-grained level. In such situations it is common for the cost of provenance capture to equal or exceed the cost of the recorded operation, leading to slowdowns of over 100%. For example, in the Lipstick system it is reported that operator-level provenance leads to a slowdown of 2-3x [5] while in the RAMP system, where provenance is collected at the tuple level by propagating tags through a Map-Reduce workflow it is common to observe temporal overheads of up to 75% [110].

Spatial overheads Similar to temporal overheads, the spatial overheads of systems recording process execution are a function of the amount of data per operation *and* the number of recorded operations. In the set of studied systems we note that only half (SPROV, BURRITO, Lipstick and RAMP) are capable of recording data changes.

While the actual overheads of any workload are sensitive to multiple factors, two reported data points are illustrative: (i) the general purpose PASSv2 system requires, on average approximately 20% additional space overhead (as compared to the original output size) to log all the operations for workloads representative of real-world applications [102] and (ii) the BURRITO system, running on a real user workload, required 800MB for provenance storage and 2GB for file versions over a two month period [68]. These results lead us to believe storage overheads should not be prohibitive for most common cases.

In dealing with space overheads, the current thesis presents an API that offers developers and system administrators the ability to configure the level of aggregation when capturing computational provenance (Chapter 3), at the expense of not being able to answer particular queries in the future. This is reasonable as the aggregation level can be set dynamically, so one can trigger finer-grained capture when detecting signs of unexpected application behaviour.

Overhead trade-offs Generally speaking, there is a direct relationship between finer capture granularities and provenance overhead. Some systems leverage this relationship to trade-off granularity of capture for provenance information. SPADEv2, for example, allows users to capture information at the function call or an application-defined level at the cost of increased temporal and spatial capture overhead. Similarly, SPROV allows users to specify modifications in higher-level semantics (e.g. “new section added to file”) at the cost of reduced per-operation observability.

In order for users to adopt the most suitable system for their needs it may be useful for them to predetermine what provenance information will be required to answer provenance queries and at what granularity this information will be sufficient, mapping it to the appropriate system.

Most systems also delay provenance construction in order to minimise capture overheads. PASSv2 for example captures raw operation records, converting them to their final representation via an asynchronous user-space daemon [102]. SPADEv2 uses separate provenance collection threads to extract, filter and commit operations to the provenance log. Other systems delay provenance collection to query time in order to avoid wasting resources computing provenance that will never be accessed. For example, Lipstick only carries out provenance construction when a query is made [5]. This delayed provenance construction property is present in some workflow systems as well. ZOOM, for example, will compute some of the provenance at query time, based on the current user view. Depending on the required cardinality, timeliness, and complexity of provenance queries, deciding on those trade-offs may considerably improve overheads.

2.1.5 Security issues

Given the amount of information stored by capturing provenance and our desire to rely on it for making assumptions about pieces of data, as well as for taking semi-automated decisions with respect to system configuration, the security of provenance data becomes a fundamental issue. As a simple example, talking about capturing provenance data in virtualized environments can only be done while addressing legitimate concerns on whether that can leak sensitive information across different tennants (virtual machines) [120]. It is also imperative that provenance data does not leak any information about the data against which it is collected [24] towards unauthorized users.

Fundamentally, this concern requires provenance data to be managed under separate access policies than the data it represents. Doing so allows flexibility over the disclosure of provenance information. For example, one might make provenance inaccessible to people outside an organisation, as it would reveal proprietary workflows or processes. However, the final data result might be freely available to anyone.

Formally, we define the security aspects of provenance as its *confidentiality*, i.e., that only authorised parties can read it and its *integrity*, i.e., that it cannot be forged or altered. We consider provenance that has both properties as essential for performing integrity, validation and consistency checks on data. The third classical notion, of *availability*, is a property of the systems collecting and managing the storage of provenance - and should be considered in that context (ensuring replication, failover and disaster recovery solutions).

Two solutions address the problem of providing secure provenance. The first leverages the concept of reference monitors: McDaniel discusses a secure system for end-to-end provenance based on the principle of a host based tamper-proof provenance monitor that mirrors the well known reference monitor concept for the enforcement of security policies [95]. The presence of the reference monitor means that the security of provenance collection doesn't have to rely on the integrity of other system components such as the kernel. While this solution is feasible, no practical implementations have been developed to-date.

The second is based on provenance chains [59, 72] where processes that generate provenance *must* attest to the information added in an encrypted, non-modifiable and non-repudiable manner. These three properties ensure that all collected provenance has the confidentiality and integrity properties. In our set of studied systems, SPROV [72] is a practical implementation of provenance chains. It primarily provides confidentiality and integrity guarantees for file modifications.

SPROV leverages a number of concepts in cryptography to fulfil the security requirements: confidentiality is maintained by encrypting the metadata describing each change, record integrity is maintained by checksumming records and attestation is supported by signing records with the public key of the creating user.

In addition to the key concepts of confidentiality and integrity, SPROV provides a number of useful features that may be of interest to the practitioner (and a consideration for future secure provenance systems): through the use of of cryptographic commitments [17], SPROV enables selective exposure of records to third parties; by employing broadcast encryption [70] SPROV supports selective access control for multiple auditors without requiring a corresponding proportional increase in the number of keys. Finally, threshold encryption [128] is supported enabling separation-of-duty scenarios in which the decryption of records requires participation from at least one auditor in a number of distinct groups.

SPROV has no mechanism for preventing unauthorised reads, relying instead on the fact that records are encrypted to prevent unauthorised access. However it is the only system in our studied set that provides any provenance confidentiality and integrity guarantees. While all systems acknowledge that the security of provenance is a fundamental concern the rest rely on existing access control mechanisms such as SQL grant privileges and file permissions to ensure security.

Table 2.1: Overview of the features and properties of the various provenance systems mentioned in this chapter. Resourceful (RSCFL) is the system I propose in Chapter 3

Type	System	Capture	Granularity	Layering	Inter-layer mappings	Versioning	Setup and Integration	Query	Time Overhead	Space Overhead	Query Overhead	Graph Construction	Security
OS / general	PASSv2	observed and disclosed[a]	syscalls; user-defined[a]	app[a], OS, storage	using DPAPI or CPL	events	custom Linux kernel	exploratory (PQL)	1% - 23%	20%	N/A	background thread	–
	SPADE	provenance providers	file-level, custom[b]	app, OS, middle-ware	fusion and composition filters	events	java app, providers	directed, limited exploratory	10%	in the order of IO ops	N/A	background thread	–
	Arnold	record, instrumented replay (R&R)	replay group	whole stack	linkage functions (pin tools)	versioning of data / app state	custom libc, pin, R&R	directed; forward/ backward	15%	< 2.6Gb/day	0.1x - 3x runtime	on query (replay)	–
	RSCFL[d]	observed and disclosed	activity, kernel subsys	app, OS, hypervi-sor	app tokens	–	kernel module, link with library	directed, exploratory (API)	1% - 20%	user-configurable	minimal (zero-copy API)	app-guided (API)	–
Workflow	Vistrails	disclosed, observed (wf evolution)	data artifacts	workflow exec, data	–	events, data ver-sioning	user defined workflows, annotations	directed, exploratory (vtPQL)	1%	minimal	N/A	on demand	–
	ZOOM	disclosed	data artifacts	application	–	events, append only fs[c]	user defined workflows, annotations	hybrid	1%	minimal	seconds	on demand	–

$\underbrace{\hspace{12em}}_{\text{Overheads}^{*}}$

Legend: N/A: Data not measured, – : Not studied / Feature not available; Continued on next page

Type	System	Capture	Granularity	Layering	Inter-layer mappings	Versioning	Setup and Integration	Query	Time Overhead	Space Overhead	Query Overhead	Graph Construction	Security
Application level	Sprov	observed	libc calls	application	–	events	linking with library	directed	11%	in the order of IO ops	N/A	–	integrity / confidentiality
	Burrito	observed and disclosed (annot)	syscalls, custom (plugins)	app, OS	using annotations	events, versioning	systemtap, plugins, user annot.	exploratory, (activity feed)	12%	2Gb/month	N/A	–	–
	CPL	disclosed	user-defined	application	manual	events	API, code changes	API	N/A	N/A	N/A	synchronous	–
Big data	Lipstick	disclosed	modules / operators / state	pig workflows	–	append only fs[c]	augmenting operators	directed	3x	N/A	seconds	post-processing	–
	RAMP	observed	record level	map-reduce wf	–	append only fs[c]	minimal code changes	directed	20% - 75%	in the order of nr records	N/A	background thread	–

$\underbrace{\text{Time Overhead} \quad \text{Space Overhead} \quad \text{Query Overhead}}_{\text{Overheads}^{*}}$

Legend: N/A: Data not measured, – : Not studied / Feature not available

[a] (PASSv2) using DPAPI [102], or CPL (Core Provenance Library) [89], disclosed provenance APIs
[b] (SPADE) given custom provenance provider modules
[c] Data versioning not necessary (all data is kept in append-only file system)
[d] This is the system presented in Chapter 3 of this thesis, for comparison. Unlike other systems in the table, it focuses on computational provenance (trying to explain how and why application properties such as performance vary or appear in the first place (errors))
[*] Typical overheads as described in the evaluation section of the corresponding system; *not directly comparable* between systems as they were not obtained by running the same workloads

2.2 Provenance in the context of system research

While so far I have described the general landscape as far as current research on provenance is concerned, it is important to also place this in the context of related research in the systems field, as there is significant overlap with previous research areas, and at the same time potential for cross-use of ideas and primitives.

versioning filesystems

The notion of keeping track of the history of data has been explored in versioning filesystems from the early File-11 [94, p53, p255] (on OpenVMS) to more current ones such as Tux3 [140]. Filesystems supporting copy-on-write such as ZFS and Btrfs also allow for the implementation of versions but are not optimised for their continuous tracking. On the provenance side, filesystems have been explored as a place of transparent metadata capture, in projects such as FiPS [136], ProFS, and more recently in a distributed filesystem context with FusionFS+SPADE [131]. Naturally, all approaches target files as the primary entity accumulating provenance and do not deal with the tracking of actual context in which content transformations happen.

version control systems

VCS solutions such as git, Mercurial or Subversion [108] gather similar information but are fundamentally targeted at direct user interaction for defining versions and retaining relevant metadata. Some prototypes even exist to turn such solutions into versioned filesystems (gitfs is one example).

Of course, the most important notion that such systems promote is one of *the (file) version* as a central element that humans think about when interacting with computing systems over time. Provenance systems need to expose similar abstractions in order to match user assumptions and intuitions.

information flow tracking

Following the history of data transformations beyond file boundaries, information flow tracking techniques have a central goal in common with provenance capture systems: tracking fine-grained metadata on process execution. This is typically achieved by applying tags to data sources and propagating them depending on the semantics of executed computations towards data sinks.

Indeed, the ideal provenance capture system would perform detailed information flow tracking on a multi-core machine, across all executing processes and inside the OS kernel. The hard part is, of course, in being able to do that precisely (without false-positive information flows) and with minimal overhead. Taint-tracking techniques [39, 152, 40] which would be applicable in principle as information flow tracking solutions, have significant overheads – between 2x and 10x depending on workload, whether they are implemented in hardware, software or need to rely on whole-system emulation. Furthermore, they suffer from over-tainting as a problem: if data flows are marked conservatively, there is a real risk of marking a large fraction of objects (sources and sinks) with the same tag. This may effectively hide the actual information flow in noise and is directly related to the n-by-m problem in the provenance area.

There have been proposals to reduce the impact of overheads by executing information flow tracking operations on a separate processor [104], by using a dynamic binary translator on given code to generate helper threads that perform the actual tracking and are kept in sync with the main execution. Similar approaches are expected to be leveraged in the provenance space.

record and replay

Of course, in light of reproducibility being seen as an important use-case of provenance, it is feasible to use record and replay techniques directly: instead of tracking explicit information flow, tracking and recording all the sources of non-determinism in a program's execution (inputs, thread interleaving, return values on non-deterministic or time-dependent functions, etc.). Such solutions exist [150, 149], although their application in SMP (Symmetric Multiprocessor) environments is challenging to implement efficiently due to the inherent multitude of sources of non-deterministic events [49, 19]. It is still possible to optimise for the SMP case by changing the multithreading strategy: instead of running multiple threads across the available processors, running multiple time-slices of the same thread ("epochs"), starting from different snapshot states. This avoids the need of recording the order of reads from memory shared across multiple threads [142]. Furthermore, when willing to sacrifice accurate replay of data races, it has been proven that record and replay can be done with overheads that allow always-on recording [92].

Those approaches can be used in performing record and *instrumented* replay [73, 46], where the instrumented part is responsible for collecting provenance with reduced overhead (in case of GUI applications, a significant fraction of the time is spent waiting for user input, but this can be replayed without delay).

reproducible builds

Finally, the approach of making things reproducible by actively *elliminating* as much non-deterministic elements as possible should be mentioned. In this context, achieving reproducible builds for software artifacts [18] has the stated goal of building a verifiable (provenance) path from source code to binary code, and allowing multiple parties to generate the exact same binary (output) by having access to the build definition (provenance) and the source code (inputs). This can be taken as an example of modifying systems and practices to allow for easier recording of provenance.

2.3 Computational provenance ties to system research

So far, I have discussed about topics in system research directly related to the provenance field. However, because the current thesis is focused on using provenance ideas in understanding more about application performance variation under contention and in allowing resource consumption attributions for debugging and diagnosis, it is important to also take a look at previous work in this space.

The issue of trying to figure out what went wrong in a complex system or why certain of its properties are not within expected intervals is not a new one. Numerous solutions have been proposed for doing program instrumentation [86, 96], tracing [132, 52], profiling [78, 74] (as mechanisms) and monitoring [1], optimisation [37, 88], anomaly detection [141] and root cause analysis [107] (for providing actionable results). Approaching the problem from the computational provenance perspective is not about reinventing such existing tools, but about understanding whether there are particular (hard) problems that would benefit from a detailed understanding of the relationships between parts of a system (provided by provenance data), in terms of faster issue identification, possibility of automation, expertise required from the operation engineers, etc.

binary instrumentation

At the lowest level, a number of techniques allow for the interposition of functions in existing binary code, without re-compilation. Such techniques rely on inserting traps or machine code trampolines calling into user-provided functions that have access to function arguments and return values. Solutions like DynamoRIO [53] or Pin [87] provide generic frameworks for applying such instrumentation across architectures. Likewise, APIs such as Dyninst [147, 118] have been developed to facilitate the programmatic analysis and modification of other binaries. The mechanisms employed are essential for building runtime tracing and profiling tools, as they allow the execution details of arbitrary programs to be captured and analysed.

Sometimes, the manual adjustment of generated interposition code is required to do instrumentations for a particular purpose optimally. The same general techniques (such as trampolines [75]) are used, but their actual machine code form is optimised for minimal architectural impact (minimising cache pollution, number of extra jumps, loads and stores, etc). To that end, in Chapter 3, I define and evaluate an optimised *call-site* interposition primitive for x86 architectures. This was custom-built to serve the requirements of collecting fine-grained computational provenance measurements (tens of thousands of active probing sites).

tracing tools

Typically built on top of primitives that do binary instrumentation, existing OS-layer trace capturing tools such as `ftrace`, `strace`, `LTTng`, `SystemTap` [84] or `DTrace` [28] mostly target batch-processing scenarios, where data analysis and event reconciliation across multiple applications is made in post-processing. Therefore, monitoring tools or schedulers only have access to *aggregated* historic data in order to make decisions about corrective actions or resource allocation strategies. Diagnosing issues starting from the same data will require human insights about where the problem lies and what probing points need to be activated. This allows for the development of tools focused on dealing with particular known issues, but leaves open the problem of figuring out similar corrective strategies in the general case. This is precisely the case for both DTrace and SystemTap, which require programmers to write small scripts in a C-like language (called D[2] in the case of DTrace), defining what parts of the OS or of applications should be traced using runtime code patching and what data should be collected. To this end, initiatives such as DTraceToolkit release lists of community-maintained scripts aimed at investigating various problems (detecting I/O subsystem, networking or scheduler issues, etc).

distributed tracing

On the side of distributed systems, increased complexity has required tracing systems such as XTrace [55] or Dapper [132] to introduce IDs that are propagated to link various parts of a distributed trace. Under my definition, those would be considered forms of computational provenance. The system proposed in this thesis, Resourceful, takes a similar approach even on a local host by introducing the notion of activity tokens. They are used for mapping fine-grained OS component measurements to individual application activities.

Another example, Fay [52] proposes a vertically-integrated approach, dealing with the collection, processing and analysis of execution traces. Its most powerful characteristic is the ability of executing SQL-like queries over such traces, considering predicates that may include references to data on multiple machines in a cluster. In that sense, it allows developers and operation engineers to investigate almost any aspect of a system and provides efficient mechanisms for transporting trace data across machines and querying it. However, its use in diagnosis is limited by what the users already know about a system: debugging a problem with Fay remains an iterative process based on insights about why the system behaves differently. This should be compared with the ideal of a guided or even fully automatic diagnosis.

performance analysis

Doing the performance analysis of a system can either rely on simulation (using a mathematical model like the ones described by queueing theory [121, 13] or used in system modelling [80, 145]), on actual measurements using one of the methods and tools described above or on a combination of the two. The advantage of mathematical models lies in their predictive power: *as long as the model is accurate*, advanced types of root-cause analysis and bottleneck identification strategies can be employed for optimising the modelled system. Of course, the difficulty is exactly in maintaining accurate models for complex, loosely coupled, systems.

CloudScope [34] is a representative example: The strategy employed aims to determine the interaction effects between different workloads in a system, based on Markov chains that model the sharing of resources. This works well for applications in which data processing closely follows well known queueing strategies (FIFO, LIFO, Processor Sharing). However, it fails for most applications based on event loops, as complex interactions between elements in the same queue are poorly modelled.

On the other hand, making inferences based solely on measured data requires significant expertise in diagnosing performance problems and an in-depth understanding of system behaviour under given workloads. More typically, the expertise of others is used through diagnosis recipes or methods (such as the USE method [65]). Alternatively, generic mathematical models can be applied and their parameters tuned to better reflect measured values [67]. Models themselves can also be inferred/learned from measurements. In Chapter 4, I propose a hybrid approach, where it is sufficient for experts to model the *relationships* that exist *between measurements*, at a coarse grained level, for being able to perform advanced analysis and prediction.

In cases where there are natural constraints imposed on a system's execution (for example, all applications executing their tasks through the use of a given RPC library or runtime), performance analysis can be conducted through instrumentation that determines the critical execution paths [54]. An operation (computation stage, function call) is defined to be part of a critical path

[2]This is **not** the same language as `https://dlang.org/`

for a given application if by reducing or optimising its execution time one would also reduce the overall runtime of the application. The same strategy can be applied, for example, to web services that are defined as workflows (in a language such as BPEL), by instrumenting the workflow execution engine. There is prior work [91] predicting performance bounds on the response times and throughputs of those services based on the workflow structure.

The aim of computational provenance-based methods developed in this thesis is to allow similar inferences on any set of services, not necessarily constrained to pre-defined interactions. I also hypothesise that it is essential to enable questions of hypothetical scenarios: taking existing application measurements and asking a diagnosis system to infer how certain parameters would look under a different set of initial conditions (configuration parameters, component latencies or throughputs, network delays, etc). The first steps towards this goal (under a set of simplifying assumptions) have been taken by Ostrovsky [107], but a more rigorous methodology is required (I explore this in Chapter 4).

In the area of automated tools for understanding system performance, Magpie [9] stands out as one of the most complete solutions. Magpie considers a high level activity (such as processing data for a request across multiple machines) and identifies instances of it from a set of system traces and measurements based on a user-provided schema. This schema of events describes the different stages through which the activity passes, but needs to be provided as user input (and is thus fixed). Combined with the fact that merging various traces in order to reconcile events across OS boundaries is done as a post-processing step, this limits the use cases in which the toolchain can be used. The proposed targeted areas are workload characterisation and system modelling.

virtualization

It is even more challenging to do performance analysis in virtualized environments [139] and to attribute performance variation to the components that are responsible for it. Work has been done in characterising the performance isolation properties of various virtualization solutions [93] ans well as on enforcing them [69]. Research on the impact of individual virtualization components such as the scheduler [35] or IO subsystem [116] have been considered before. In Chapter 4, I will approach this issue from a different perspective that shows the value of computational provenance: can one determine how much are individual application activities (i.e. servicing a http request) slowed down by virtualization and contention in cloud environments?

performance interference

Virtualization adds more opportunities of performance interference through the co-localisation of multiple VMs or containers. Interference can also occur locally between applications contending for the same resources (CPU, caches, memory, I/O). Interference is one of the behaviours that is challenging both to model and measure [143], being an active topic of research. Previous work has looked at interference happening in virtualized environments [79, 105], caused by hypervisor scheduling [151], by network contention [117] or storage I/O [30]. It is of course hoped that identifying interference would lead to systems that are aware of it and adapt their scheduling (of CPU tasks, network access or block I/O) accordingly. However, it's worth noting that such adaptive models are yet even harder to model, debug and reason about in terms of performance. In this thesis, I am tangentially describing interference problems, from a different perspective: can *changes* in the performance of fine-grained activities (i.e increase in latency for a web request) be attributed to particular components in the system? When attribution is exact, the causes of slowdowns could be concurrent applications contending for the same resources and causing interference.

root cause analysis

In that sense, interference sometimes presents itself as a problem solvable through root cause analysis [106]. However, root cause identification is a more general problem, also looking at determining causes of errors, failure and incorrect behaviour besides performance degradation. This means that in general the "root causes“ can also point to problematic configurations or inputs. X-ray [6] for example looks at estimating the cost induced by certain configuration values on the execution of basic application blocks. The mechanism proposed (based on information flow tracking and binary instrumentation) is also able to explain the differences observed between the execution of two similar activities.

Lower overhead solutions are possible for diagnosing the root cause of performance variations in communicating services [2, 107]. Similarly, bottleneck identification algorithms have been proposed as primitives for identifying root causes for services running in the cloud [77]. The approach developed in Chapter 4 complements those, enabling the construction of predictive models rather than just the diagnosis of existing situations.

"Any fool can know. The point is to understand."
— Albert Einstein

3 *Resourceful*: Placing kernel measurements in context

AS COMPUTER SYSTEMS increase in complexity, it becomes harder to understand and control their behaviour in terms of performance, resource consumption, failure modes or response to changes in configuration options. To take an example, we can look at the software running on the current infrastructure of large Internet companies such as Google, Microsoft or Amazon. In order to respond quickly and reliably to high volumes of client queries, their services are built as multi-tier architectures, with each tier (e.g front end, caching, application, database, storage backend) being separately provisioned, replicated and/or sharded. The processes implementing each tier's functionality are then typically run inside virtualized environments (either containers or VMs).

Cloud computing services are making the building-blocks of such scalable architectures widely available. The expansion of mobile services and ultimately IoT will further drive the adoption of similar distributed and virtualized architectures across the industry, with the additional disadvantage of not having a single entity to manage the end-to-end operation of services. For example, this raises the question of how end clients can depend on services provided by companies which in turn use a chain of other providers for things like compute power, storage, data analytics, configuration management, etc.

Because of the large number of moving parts and configuration possibilities, it will no longer be sufficient to monitor systems and provide alerts when certain metrics hint at anomalous or suboptimal behaviour. Without in-depth knowledge of how every part of the system works, as well as knowledge of the overlapping workloads running on the same physical servers in a shared infrastructure, it will become challenging to return the system to a working state after failure and even to understand what components or service providers are to blame. *However, it is preferable that such knowledge is not required from the people operating those services, but instead held by automated systems.*

The work presented in this chapter provides primitives for dealing with such issues, when considering data about the behaviour and properties of systems at different points in the software stack. By capturing the *context* of every measurement made, as well as the relationships between various metrics, the proposed solution targets the ability to trace problems back to the components that have triggered them. Seen from a high-level perspective, this is a type of provenance that deals with capturing the various factors affecting the properties of final outputs (e.g. latency, number of errors, dependence on configuration parameters), in a way that allows explanation and attribution (Note 3.1).

The starting hypothesis is that by capturing this provenance data, a digital assistant can be built to help people understand where things have gone wrong in a complex system, reducing the amount of in-depth knowledge needed for taking corrective steps as well as reducing the time-to-solution. In an ideal (and distant) world, such an assistant will simply determine what's wrong with the system and then autonomously take corrective actions for keeping it running optimally.

Note 3.1: Provenance vs logging, instrumentation and monitoring

While precisely defining boundaries between terms such as logging, instrumentation or monitoring is somewhat arbitrary because of their overlapping meanings, I will consider them as follows:

logging: The action of recording data about the execution, internal state or characteristics of a process. A restricted view would consider logging as only related to *actionable* pieces of information[a] (e.g "database connection failure" or "indexing complete").

instrumentation: Pieces of code responsible for capturing (measuring) the state of a process or its performance at a given point in time.

monitoring: The use of historic measurements about the state and performance of a system for assuring its continuous operation within acceptable parameters.

Distinguishing between provenance data and what is normally obtained from logging and performance measurement tasks is a matter of perspective: the latter are typically obtained in a general setting and aggregated (measuring performance parameters for a system in order to do monitoring), while what I label as provenance ties any obtained data to particular outputs/activities of the application. To give a simple example, one generally thinks about latency in terms of probability distributions (as is the case when monitoring). However, having access to provenance data will let a developer analyse the interrelationships between components that have contributed to the latency of individual actions (like processing a user request).

Logged data contains information about the steps taken by an application or about side-effects produced at runtime. Complete provenance should explain how those recorded events influenced each other and why.

When it is possible to *infer* context and deduce exact causal relationships from a complete log of high-level events and aggregated measurements, provenance is nothing more than an index into that set of events, revealing the underlying relationships.

[a]based on a definition by Peter Bourgon

Some of the complex issues that appear in systems like the ones described above are difficult to diagnose because of unwanted side-effects and abstractions hiding away the details of how resources are shared amongst multiple entities: an application will typically have the illusion that it's running alone on a physical machine. However, that is a leaky abstraction: the application's performance will be affected by concurrent workloads and virtualization, while some of it's failures will depend on the failure modes of shared resources (network timeouts, full buffers, hardware errors). It is also unclear how local changes (configuration, updates) might affect the system as a whole.

The framework presented in this chapter, Resourceful, provides the primitives for solving such issues by gathering provenance measurements from precisely the places responsible for resource allocation and sharing: OS kernels and hypervisors. The overall design focuses on three significant motivational use-cases:

Understanding performance: In complex web application architectures, answering a single user request could involve processing taking place on thousands of servers (fan-out architectures). This means that even relatively rare performance hiccups at the level of individual servers may translate into poor average/median performance from the perspective of the end-user [45]. Note 3.2 explains this behaviour further. However, when running inside virtualized environments, a trade-off exists between performance consistency and utilisation: higher server utilisation (and a more efficient use of available resources), also typically increase the likelihood of component-level variability [10].

Solutions to this issue have been proposed, but most of them are focused on hiding the variability rather than understanding why it is there in the first place. For example, a well known approach is to issue multiple requests to the system and only return the answers of the fastest response, while cancelling others [44]. This means that one can obtain both better performance consistency and server utilisation, but at the expense of doing needless work (hence, part of the increased server utilisation is artificial: resources were merely spent on processing requests which are then cancelled). Ideally, we should apply such solutions only after knowing that the observed variations in performance are unavoidable and can't be fixed through system optimisations.

This leads us to a first guiding design principle: for understanding performance variability, it is not sufficient to take snapshots of various system metrics, as typical logging and monitoring tools do; instead, we need to capture enough information to understand what factors have contributed to the observed values.

For example, on high-latency requests, there is a need of diagnosing causes: What is different from the low-latency case? Were there unintended interactions between the server and other co-located applications? Where was most of the time spent? Are there configuration options which when changed might improve the end result?

Note 3.2: Details on the impact of high fan-out on end-to-end latency

This back-of-the-envelope calculation considers a front-end sending 500 *independent* requests to identical "leaf" services and collecting the answers in order to compose the response. The end-to-end latency is presumed to be the maximum of those latencies.

Let $F_{U_i}(u)$ be the cumulative latency distribution functions of individual services, $i = 1..500$. Assume that for all services, we measure the 99.9^{th} percentile of the latency distributions to be $512\,\mathrm{ms}$:

$$F_{U_i}(512\,\mathrm{ms}) = P(U_i \leq 512\,\mathrm{ms}) = 0.999$$

We can now compute in what percentile would $512\,\mathrm{ms}$ fall for the end-to-end latency distribution (F_Z):

$$\begin{aligned} F_Z(512\,\mathrm{ms}) &= P(Z \leq 512\,\mathrm{ms}) = P(\max\{U_i\} \leq 512\,\mathrm{ms}) \\ &= P(U_1, U_2, ..., U_{500} \leq 512\,\mathrm{ms}) \overset{i.i.d.}{=} \prod_{i=1}^{500} F_{U_i}(512\,\mathrm{ms}) \\ &= 0.999^{500} = 0.606 \end{aligned}$$

Therefore, it turns out that for the end-to-end latency, $1 - F_Z(512\,\mathrm{ms}) = 0.394$ (39%) of the requests are slower than $512\,\mathrm{ms}$.

Comments

The results for the case described above show that if the individual "leaf" services have a 99.9^{th} percentile latency of $512\,\mathrm{ms}$, (1 response in 1000 takes $512\,\mathrm{ms}$ or longer), then in the end-to-end latency distribution, 39% of the responses are expected to take $512\,\mathrm{ms}$ or longer. In other words, even if all the individual services are well within their SLAs, the performance as experienced by end-users might still be unacceptable.

In practice, this means that slow servers/responses, even if rare, are very important to the overall perceived performance. Any effort spent in improving their latency (even taken as a non-distributed problem, of improving the local effiency of particular servers) is worthwhile.

Resource consumption: Irrespective of the underlying reasons (mandated policy or costs), there has been a constant push towards running services in a way which makes optimal use of the available resources. However, most of the understanding about what resources are used, from individual servers to whole data centers is obtained by taking aggregate measurements. Similar to the case of application performance, those aggregate measurements will allow for coarse-grained accounting but not attribution. In order to explore the opportunities for optimisation, as well as more precise end-user cost models, a transition must be made towards fine-grained contextual measurements which can then be aggregated in multiple ways: per user, per application, per activity across data center etc. For example, one might want to understand the cost of a client request, either on a single host or over a distributed system; this implies being able to aggregate data at sub-process granularity (summing over the calls that were made to service that request).

This represents a second design principle: make fine-grained measurements and their customised aggregation possible. This can be achieved by imposing low per-measurement overheads and by employing in-kernel aggregation configured through user-space API calls.

The scheduling and availability of resources (queuing, saturation), plays an important role in defining cost models, as well as end-to-end performance characteristics. While Resourceful is currently focused on understanding individual server data, its design takes into consideration the possibility of integrating the measurements into data-center scheduling decisions.

Service configuration: As some applications transition from monolithic designs to ones based on microservices, the importance of having systems where DevOps teams or end-users can understand the impact of changing various configuration options will become critical. Two elements related to provenance are very important in this use case: (i) capturing the local knowledge of how changing configuration options affects the operation of individual applications and (ii) inferring the behaviour of the system as a whole based on inter-service dependencies and the use of common resources.

Both microservice and tiered architectures face issues with understanding the full set of dependencies for a given service, and how the operation of the service is affected by changes in the configuration of those dependencies, their failures, resource sharing and saturation.

The resulting analysis should allow asking "what if?" (counterfactual) questions about the running services. However, such queries should be understood from the start to have a *statistical* rather than an exact meaning: the effects of configuration options that were never explored will still be impossible to determine unless somebody provides a regression model that can relate their current values to other known configurations.

The idea put forward in this thesis and justified over the next two chapters is that computational provenance constitutes a channel for obtaining in-depth knowledge of the system state, of its performance and for predicting potential side-effects of changes in configuration, updates or failure states. The longer-term view is one of using this data as part of feedback loops in semi-automated diagnosis systems.

Resourceful, as well as the tools built on top of it, take the first steps in that direction, with a focus on offering insights about failures, observed variations in performance under various system configurations and resource consumption.

Note 3.3: Resourceful development and contributions

Resourceful was created, from the onset, as a project with contributions from multiple researchers. Besides myself, both Oliver R.A. Chick and James Snee had significant contributions in the design and implementation of the system. However, the current chapter focuses on my own contributions. Whenever particular components were developed as part of joint work, but still need to be presented here in order for other sections to make sense, a footnote will be present for clarification.

3.1 System Design

Resourceful is designed to give applications full control over the fine-grained measurement and aggregation of the resources they consume. It is able to provide this type of attribution while also recording data about events happening concurrently, enabling the exploration of unwanted interference effects between different workloads. This remains within the realm of computational provenance as it determines the elements influencing properties of the considered data: the fact that a web request has been serviced concurrently with 100 others will impact the latency of all of them.

This differs from current monitoring and performance profiling practices: The level of aggregation for typical monitoring systems is either at the *system operations level* – for example, targeting datacenter site reliability engineers looking at server utilisation, VM scheduling and migration or network traffic – or at *application level*, where metrics used by development teams (i.e. latency, errors per second, transaction timeouts) are collected periodically. In every case, alerts might be triggered if the measured values exceed certain thresholds for a given amount of time.

multi-level analysis and aggregation

However, despite taking place at different levels of granularity and abstraction, the two sides (dev and ops) of monitoring should not be seen as independent. Instead, monitoring should be recognised as a cross-cutting concern, where no single view of the world will fully provide ideal insights/solutions. Application metrics might be outside acceptable intervals *because* of various operation-specific conditions (queueing, hypervisor contention, network degradation or oversubscription, etc). Developers will thus need to be aware of performance metrics from the ops-managed hardware while the operation engineers will need to understand the aggregate metrics of their systems in the context of what the currently running applications are doing (seeing network and disk utilisation spike during an indexing operation is different from similar spikes caused by large user queries). This is why Resourceful chooses to put every measurement in the context of other measurements and system parameters, enabling a multi-level exploration of the collected data.

Understanding the context in which a measurement was made is important because it will highlight queuing and resource contention issues, but alone it can't explain why a particular value was observed for any given metric. To provide explanations, we also need to determine what constitutes the "critical path" for a given high-level application activity [107]. Such critical paths are defined by considering the various processing or waiting steps done for completing a high-level action. If reducing the latency of a given step reduces the latency of the whole action then that particular step is said to be part of the critical path.

data per functional subsystem

Identifying such paths at a granularity that allows meaningful optimisations is challenging: a too fine-grained approach will yield information about details such as individual kernel operations, upon which a user-space developer can not immediately act; on the other hand, a too coarse-grained recording will leave the same developer with too little data about what should be optimised. Resourceful searches for a balance by presenting the information in terms of application activities and kernel subsystems (CPU, Xen, Memory, Network, VFS, Block devices, etc). For example, the data recorded for one system call would contain total CPU cycles, wall clock time and memory costs, but these values are further broken down for each functional subsystem touched during that call: total CPU cycles spent in the network subsystem, total cycles spent in

VFS (Virtual File System) and subsystem-specific metrics such as bytes sent/received, number of retransmissions, IO queue size and disk writes. The exact set of recorded metrics is configured through an API.

self-monitoring

Another element that differentiates Resourceful's design from current approaches is its focus on 'live data'. It allows for both self-monitoring and monitoring by third-party applications, with the measurement results being available for adapting application behaviour or system scheduling operations on the fly[1]. This is a critical element for using such data as part of feedback loops. The approach is feasible only in the context of fine-grained measurements backed up by an API that allows immediate access to their values.

In contrast, most high-level interfaces to performance analysis tools (perf, ftrace, SystemTap) have a batched approach, where measurements are sampled and post-processed at a later stage. User-level processes typically need to read trace buffers before they are filled to avoid loosing events, while aggregation is either limited (SystemTap) or needs to be done manually.

Resourceful is closer to low-level interfaces such as `perf_events` – that allows programmatic access to performance counters – or to the `getrusage` system call, that allows for coarse-grained resource consumption measurements. At the same time, it provides high-level aggregation primitives that are simple to control and understand, by introducing the notion of *activity tokens*. Applications use the API to notify the kernel of changes in the current activity being executed, so that consumed resources can be attributed accordingly.

custom in-kernel aggregation

Placing this in the context of provenance, the responsibility of disclosing activities and declaring interest in resource consumption belongs to the application precisely because the per-token aggregated data is only useful in the context of application semantics. For example a web server defines the set of operations that make up "responding to a user request", and is then able to query for resources being consumed during such activities.

By reporting resource consumption hierarchically aggregated per $(token, subsystem)$, Resourceful provides a more detailed view of what happens inside the kernel. For example, given a socket `send()` operation, the application can view the breakdown of latency and answer questions such as: Was most of the time spent in the network stack? Was the packet delayed by the scheduler moving the task on a different core? The application can then classify its own service times by comparing requests that have significantly different performance characteristics.

The basic building block for answering those questions in practice is *activity tracing*, which I examine more closely next. This provides a justification for the way Resourceful collects measurement data.

Activity tracing

The main obstacle in accurately diagnosing how time is being spent inside an application and understanding what needs to be optimised is doing measurements that ignore hardware, OS and application resource multiplexing.

Such measurements are typically performed by starting and stopping counters (or reading performance counters) at the beginning and at the end of a high-level operation, and then computing the delta time (or event count) between the two snapshots. If any multiplexing took place[2], the measurement will attribute a fraction of the observed value incorrectly [8]. Furthermore, the amount by which the measurement is wrong increases with the duration of the measurement, as more unaccounted-for multiplexing takes place (typical for application or OS level measurements). A concrete example of this follows.

[1] I am collaborating with the University of Cambridge's High Performance Computing Service for implementing such a solution on top of Resourceful

[2] if the operation is non-trivial, it is more than likely that such multiplexing took place, at some level (OS kernel, virtualization, hardware), unless specialized configurations are in place (i.e. CPU pinning, setting interrupt affinity, avoiding event loops)

From the perspective of a provenance system, this is an instance of the n-by-m problem: not being able to distinguish between individual contributions to the measured values leads to difficulties in precise diagnosis and debugging: while the end result will still correctly reflect how long it took to perform the operation, it will most often fail in answering *why*.

To highlight this issue, Figure 3.1 presents a small time slice of fine-grained measurements taken from the `lighttpd` web server while processing http requests. Each timeline (different values on the y axis) represents the work done for a particular server connection object.

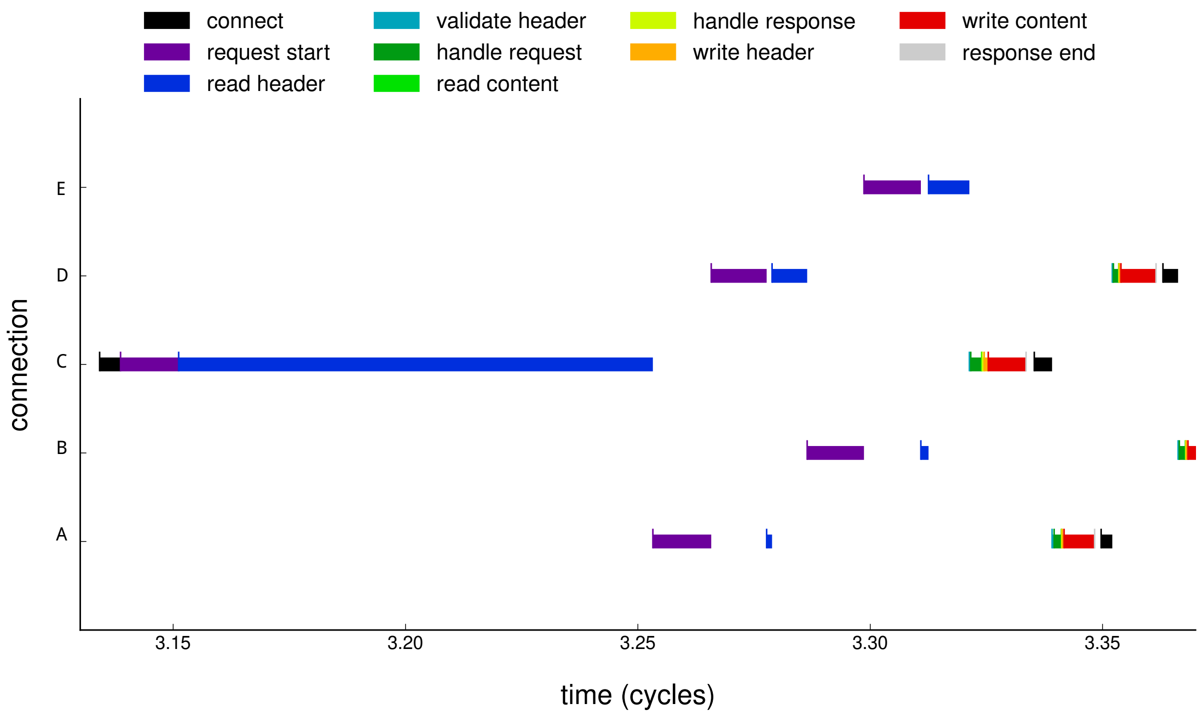


Figure 3.1: lighttpd - Event loop request multiplexing

Multiple connections are "active" at a time, and `lighttpd` multiplexes between them using an event loop [3]. After a response has been sent, the state of the connection is reset and further client requests can be served using the same underlying object.

In this case, using queuing theory for gaining insights into how the final latency distribution changes in response to changes in various other request or application parameters is not straightforward: the processing is done in multiple stages and using a common queue, with the latency of one request depending on the latency of requests that arrived after it in the queue. It is therefore expected that M/G/1-PS[4] queuing models, such as the ones used by CloudScope [34], would not be accurate for event-loop based applications. Resourceful considers an approach based on real-time measurements instead, on top of which more realistic models can be built.

To better describe the issues encountered with current strategies, assume coarse-grained latency measurements for the first client requests served by connections C and D in Figure 3.1, using the counter start/stop strategy: this means we'll take four counter snapshots and end up with two intervals (between the start of the black intervals and the end of the red ones). From the application's timeline point of view, some intervals are accounted for multiple times (the parts for which the requests were waiting for each-other or on other connections) as can be seen in

[3]Here, lighttpd is running on an isolated CPU and no OS scheduling takes place: if one would project all intervals onto a single axis, a continuous colored region would be formed

[4]Markovian arrivals, Generalized distribution of processing times, with 1 server and a Processor Sharing service policy

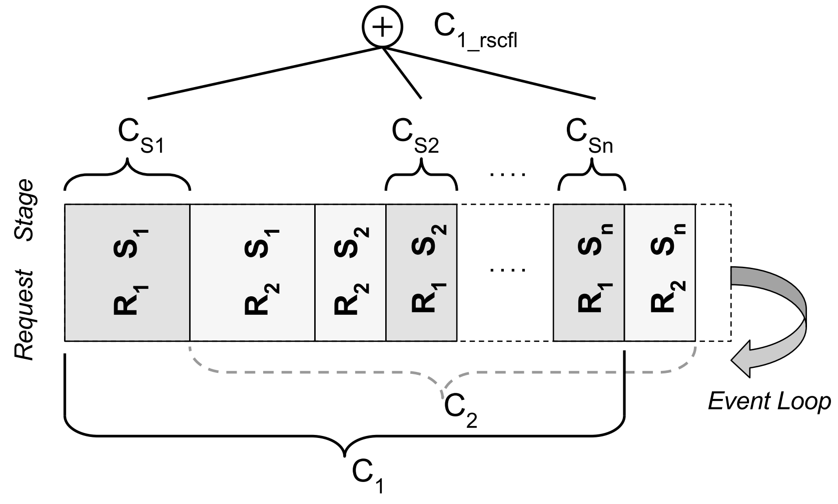


Figure 3.2: Comparing Resourceful activity measurements (above) with naive request start/stop performance counter snapshotting (below) in the case of `lighttpd`. The web server processes requests in multiple stages (HTTP header parsing, file retrieval, sending response) and schedules the processing for those stages on a single thread using a simple event loop.

Figure 3.2.

Another way of confirming this is by summing the obtained server-side latencies and observing that their total time exceeds the actual runtime. If at the same time performance counter measurements were taken (even while limiting their scope to the server process), the counts for serving a particular request will include events that belong to other requests. In fact, the measure

$$W_s = \sum_{i=1}^{n} lat(i) - \text{process\_run\_time}$$

will represent the total waiting time (across all requests) due to user-side multiplexing, with $W_s = 0$ when no event loop multiplexing takes place (requests are served to completion before considering other connections). This can be used as a measure of intra-process contention.

However, aggregate measures like these can't provide data for comparing the latency of individual connections in order to explain their differences: were they different because one waited longer for other requests – as is the case of request D (a workload interference and queuing problem) or did particular stages in their processing took longer - as is the case of request C (something which must be explained by further time break-downs). Furthermore, beyond direct comparisons, developers would be interested in identifying different classes of requests in accordance to the characteristics that most influence their latency.

Such results can only be obtained if fine-grained measurements are taken throughout the different stages in the lifetime of a request. Figure 3.3 summarises the high level behaviour of the `lighttpd` web server as determined from Resourceful measurements. It looks at the proportion of time spent doing actual work and waiting for resources or other requests, ordering the requests on the x axis from low (on the left) to high latencies (towards the right).

From the figure, we can identify 3 interesting areas: for the low-latency requests on the left, most of the time is spent doing actual work; the proportion of time spent doing useful work drops quickly with most mid-level latencies (between request `2 000` and `6 000`) being dominated by wait times. Reducing those latencies will require altering the application-level queuing and multiplexing strategy. In the third area (tail latency) we observe a bigger split between requests with similar final latencies but with different causes for that latency: it is clear that some requests take longer because of application level multiplexing while others spend unusual amounts of time

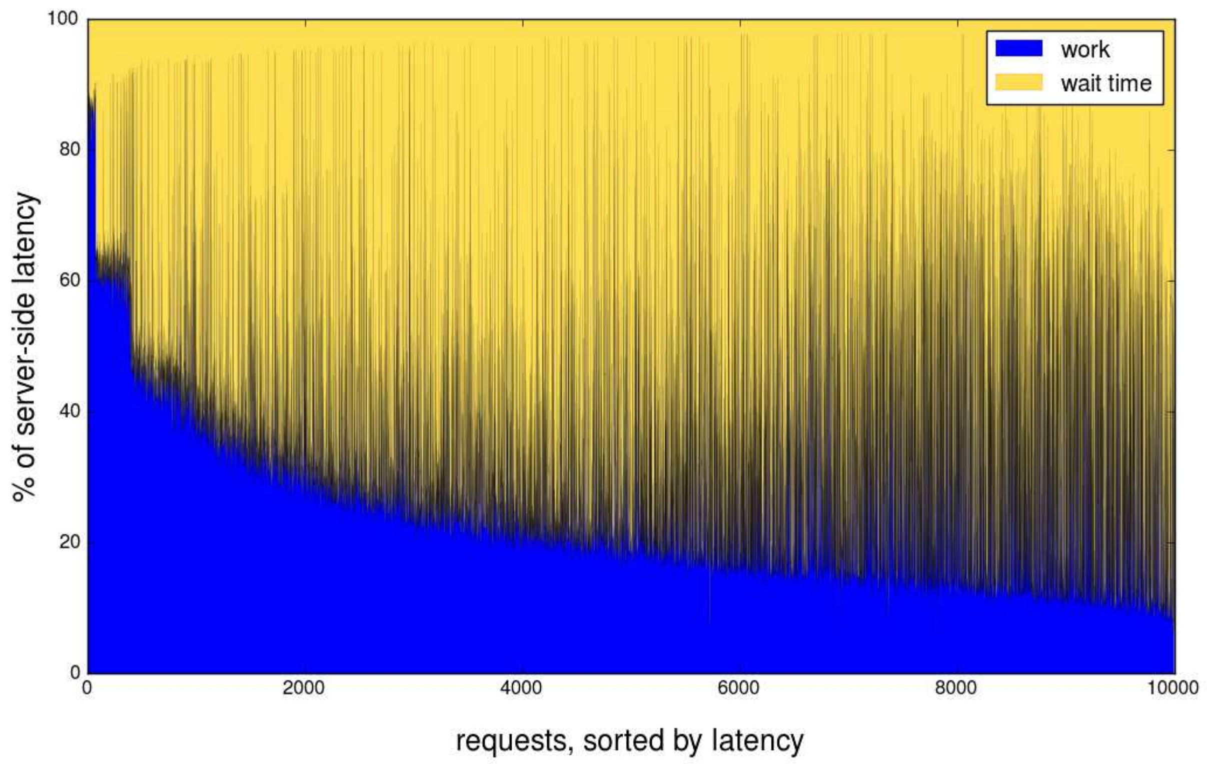


Figure 3.3: lighttpd - latency breakdown highlighting the percentages of work (time spent doing actual processing) and wait times due to application-side request multiplexing. Each vertical line represents one request, and is divided into two parts representing the percentages of work and waiting. The requests are ordered from low server-side latency (on the left) to high server side latency (on the right). For high latencies, we observe two types of requests: the ones which wait for a very long time (more than 80% of the service time) and the ones for which the actual processing took a long time. Not being able to differentiate between the two groups would make latency analysis difficult.

doing work.

To understand the causes of tail latency, we will have to isolate this last group and drill down into the finer-grained kernel-level information provided by Resourceful for determining possible optimisation strategies. Reducing the latency for those requests will also improve the latency of the other tail-latency group: those were the requests which have waited for longer because of atypical work done by other requests.

The discussion above justifies our aim for accurately tracking application-level activities. With increased interest in event loop-based applications, the described multiplexing strategies are becoming more common. This is especially true in systems designed for high performance and scalability, where there is a clear need for understanding behaviour beyond what traditional queuing theory models offer.

Resourceful's design decision in regards to this is to provide API functions that applications can use to announce whenever they switch from one activity to another. On starting a new logical activity, the application needs to call `rscfl_get_token`, receiving a new opaque token object. This object needs to be passed to `rscfl_swap_token` or to other resourceful per-system call accounting functions whenever work is being done for that activity.

While doing this might be trivial in some applications (`lighttpd` required just under `50` lines of additional code), it is not always convenient or possible to modify the source code for inserting Resourceful API calls. However, it is possible to use techniques such as the Linux `LD_PRELOAD` to externally inject function wrappers implementing this functionality into existing binaries. For example, I have been discussing this for the case of HPC applications, where Resourceful can hook into the MPI_Control hooks provided by certain closed-source MPI applications for activity monitoring purposes. It is conceivable that such tasks could be automated through the use of static analysis.

External data: In order to facilitate integration with userspace measurement tools or monitoring software, Resourceful allows arbitrary application data to be associated with any token. This can be used to provide further context to kernel-side measurements, by adding application-specific metrics for each activity.

Tracking asynchronous behavior

Of course, not all resources required for completing a given activity are consumed while the application is active or when it knows to what token they should be attributed to. This is an effect of operations happening asynchronously, either in the kernel or at application level. I consider two types of such asynchronous behaviour:

1. resource consumption happening *before* we know who it should be attributed to: for example, this happens with resources consumed on the receive path of the TCP stack, before we know to what application is a particular packet addressed to. A similar situation is encountered in user-space applications using calls such as epoll: implicitly, those are used to select file descriptors with pending events; in the case of file descriptors representing socket connections, the application will not know to which of those should the time spent doing the epoll be attributed to before that function call returns. Indeed, the time should be attributed to the connections with pending events *returned* by the epoll.

2. resource consumption happening *after* an operation has completed from the perspective of an user-space application: this happens whenever buffers or write-back caches are involved, as some of the operations that can be attributed to an application activity have delayed effects: in case of a buffered write to disk, how can one split the cost of doing a flush amongst all the application activities that had produced the data?

Without measuring amortised, asynchronous costs, existing performance monitoring tools provide an incomplete representation of system resource consumption. It is currently very hard to understand how such operations affect concurrently running workloads, and it is at least plausible that they might sometimes explain bad interaction effects or performance degradations.

Resourceful's design tackles the issue depending on the type of asynchronous behaviour. For type 1 (actions happening before we know to whom we should attribute them), the solution is to provide primitives for merging the resources consumed for a given token into other tokens: for example, the application will activate a new non-action-specific token before an epoll call. After the call returns, one can merge the resources associated with this token into the measurements of other tokens using an API call (merge_acct_into).

In regards to doing measurements for type 2 events, we require the ability of tracking kernel data structures together with the operations done on them: If an application writes data to a buffer (for example, a socket writing data into a shared qdisc buffer), we must be able to identify further operations on that buffer until the data is flushed. *Part* of the costs of maintaining and flushing the buffer, together with subsequent operations on the data (i.e code inside the network card driver) can be attributed to the original writer. Detailed tracking will require the propagation of ID hierarchies in order to identify actions across layers of abstraction (for example an application waiting on an NFS file write, which in turn waits on data from a TCP connection). At the moment, Resourceful only deals with the propagation of application-level IDs (abstractly defined as tokens)

Naturally, dealing with type 2 events is potentially very expensive: it might require trapping memory accesses to particular data structures. Fortunately, the Linux kernel is typically organised in a way that encourages maintaining such shared data structures through common functions. In the case of the running example, the kernel will call the `dev_queue_xmit` function to enqueue socket data into the `qdisc` structure, while the `qdisk_restart` function will clear it. Tracking those operations using normal probing is sufficient.

Beyond measurement however, the problem of attributing parts of the resources consumed asynchronously to the activities that have triggered them is not well defined: indeed, from case to case developers might need to pick different attribution strategies: for buffered writes, a simple strategy would be to divide the asynchronous costs proportional to the size of each write. For functions such as `epoll` and when analysing latency, it makes sense to attribute the full cost (max) to each of the file descriptors that are returned: indeed, if we were using `epoll` to multiplex between multiple requests, its latency will be added to all of them. As a consequence, the attribution strategy needs to be decided through a user-provided function.

General architecture

At a high level, Resourceful it is built as a measurement framework with the following components:

1. *Measurement points analysis:* this performs a static analysis of the current kernel binary in order to identify a minimal set of instrumentation probe points and subsystem boundaries[5] (the level at which primary aggregation takes place), guided by a user-provided configuration;

2. *A kernel module* responsible for inserting custom, low-overhead measurement probes into the kernel and activating them when applications request resource consumption data; those probes contain the code that performs the actual measurements and relates them to each other.

3. *A user-space library* exposing the API that applications can use to express interest in the resource consumption of particular system calls and to read the results after the required information was gathered on the kernel side.

4. *Data management/export facilities* for applications that link against the library, in order to perform subsequent analysis. The system presented in Chapter 4, Soroban, makes use of those facilities for enabling the training of machine learning models for predicting application performance under different scenarios and answering what-if questions.

3.2 Implementation

3.2.1 Measurement points[6]

Knowing what places in the kernel need to be probed in order to obtain meaningful data from measurements, while at the same time minimising overhead is very important, especially as we aim for fine-grained activity tracking. Existing tools such as `perf` or `ftrace` report their data at the granularity of kernel functions. While some of those are interesting from a user-space perspective (i.e. methods for acquiring locks), having measurements in relation to low-level kernel functions is at an abstraction level which can't be directly mapped to user-space concepts. Trying

[5]main contribution by Oliver R.A. Chick : the core insight was using the Linux MAINTAINERS file for identifying subsystems

[6]main contribution by Oliver R.A. Chick

to obtain full stack traces that would put those measurements in context, for example by using the ftrace `function_graph` tracer, will impose non-negligible overheads while also failing to provide details about the particular activities that were affected by slow execution, transiently: was a http server processing `request_1` or `request_2` when trying to acquire a contended lock?

This means that the data can not be directly used for optimising applications or understanding why they failed. Furthermore, the Linux kernel (as of the 3.19 version) has 28 930 functions. Probing all of them would add significant overhead and would impact even applications which are not being instrumented, because of the kernel code being shared amongst all processes.[7]

Resourceful takes a different approach: it groups multiple low-level functions based on their purpose: For example, "functions dealing with IPv4 networking", "functions dealing with virtual file systems" or "functions dealing with power management". Such groupings relate directly to the Linux notion of *subsystems*. Fortunately, the kernel keeps an up-to-date list of such subsystems in its MAINTAINERS file, together with the source files belonging to each subsystem (Listing 3.1)

At build time, a static analysis of the current kernel's binary is performed, based on existing debug symbols. It maps addresses in the kernel from where function calls are made (call sites) to the source file where the function being called is defined. This information, together with the data in the MAINTAINERS file, is used to determine whether a particular function call crosses from one subsystem to another (the call site and the function definition are in separate subsystems). A header file is generated to contain those subsystem boundary addresses, where measurement probes will be placed.

```
NETWORKING [GENERAL]
F:   net/
F:   include/net/
F:   include/linux/in.h
F:   include/linux/net.h
F:   include/linux/netdevice.h

NETWORKING [IPv4/IPv6]
F:   net/ipv4/
F:   net/ipv6/
F:   include/net/ip*
F:   arch/x86/net/*
```

Listing 3.1: Fragment from a Linux kernel maintainers file

Two architectural decisions are made in respect to the points of measurement:

- doing per-subsystem measurements, which is more meaningful from the perspective of a user application.
- adding probes around *call sites* rather than inside the target function. *This means that not all calls to a given function are probed, but only those coming from another subsystem.* The number of call sites, even when just considering the ones which cross subsystem boundaries, far exceeds the number of functions. However, even though the absolute number of required probes is higher (87 000 and over 140 000 when considering function pointers as well) this limits probe effects and reduces overhead.

[7]The Shadow kernels paper [Lucian-S5], to which I have contributed as a co-author, presents a possible solution in this space

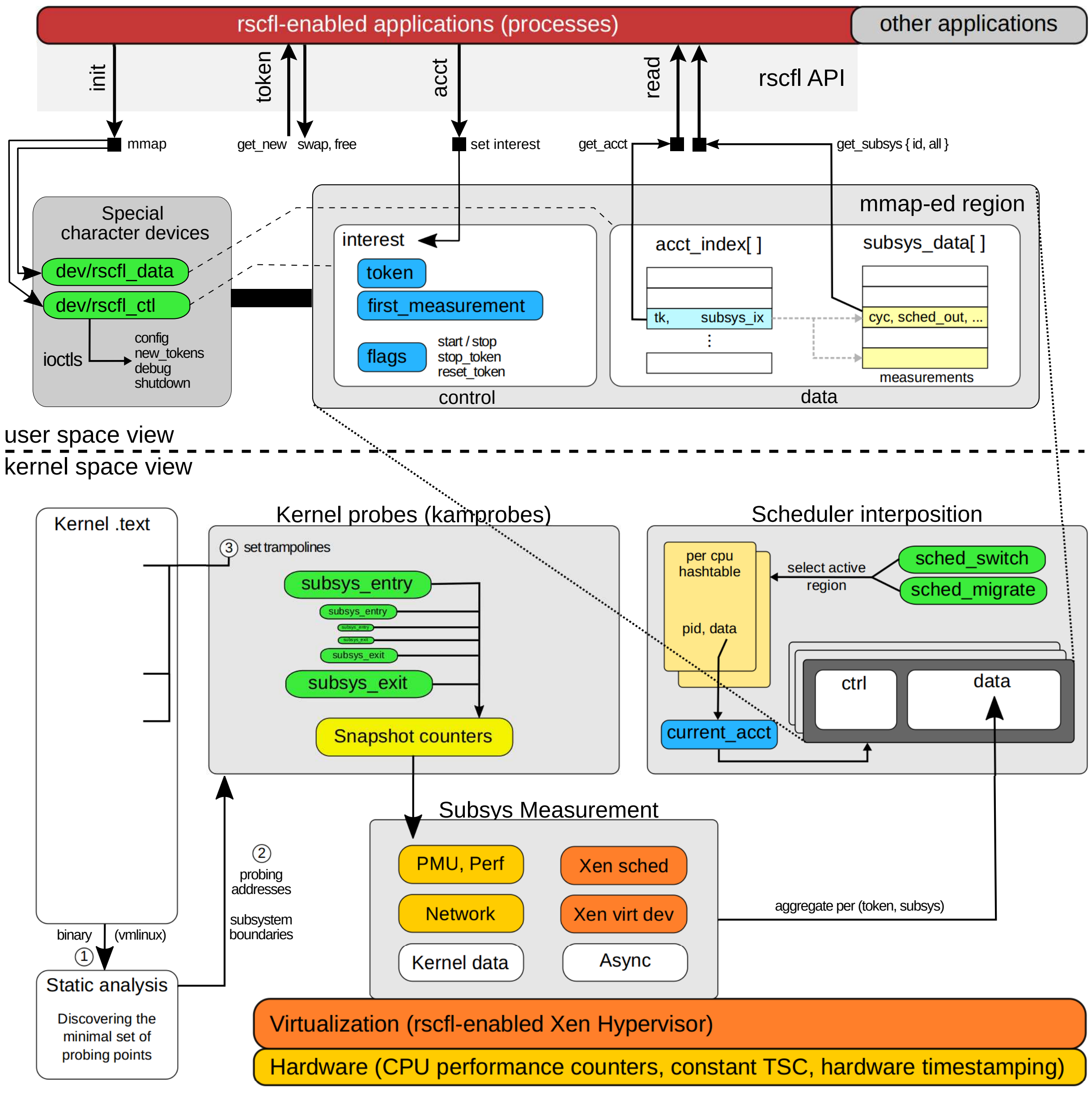


Figure 3.4: Resourceful architecture; implementation details

3.2.2 The Resourceful kernel module

A kernel module is responsible for dynamically inserting probes into a running kernel, recording provenance data about system calls that are executed and about the context in which they execute (determining what other workloads are competing for the same resources as an instrumented application).

The resulting data is stored within the kernel and aggregated according to opaque handles provided from user-space (the activity tokens). This allows applications to do custom "per-activity" resource aggregation. For example, a web server can use the same token when doing all the processing for servicing a http request.

The kernel module exposes two new character devices to user-space (Figure 3.4): one responsible for data (`/dev/rscfl-data`) and the other (`/dev/rscfl-ctrl`) responsible for control messages. When the instrumented process mmaps the data device, a memory region specific to that thread is created on the kernel side and registered with the kernel module, in a per-cpu, pid-indexed hash table. This region will hold all measurement data for the given thread and will be directly accessible to the application in its address space. Other applications (monitors) can be configured to process data on behalf of an instrumented process when needed (if they have sufficient permissions to read from the data device).

The control device holds internal state regarding what system calls should be instrumented as well as user-preference flags controlling the actual instrumentation. Furthermore, the devices exposes a number of ioctls for general communication with the module (start/stop, probe insertion, benchmarking flags or configuring what probes should currently be active) .

The low-level interaction with those character devices is abstracted away by the user-space API. Applications will use this API to get access to all Resourceful functionalities, as well as more complex user-space data processing (functional-style map-reduce features).

Architectural decisions:

- Allow all functionality to be placed within a kernel module, without requiring kernel source code changes;
- Define optimised x86 trampolines (called kamprobes) for low overhead probing, because of issues with existing kernel mechanisms (kprobes, ftrace and in part eBPF in its current form);
- In-kernel aggregation of per-activity data, while retaining per-subsystem breakdowns;
- Define scheduler interposition and per-cpu data structures in order to reduce per-probe overhead and eliminate locking

Kamprobes[8]

The main problem when attempting to systematically measure what happens in the Linux kernel is the fact that current probing mechanisms are not designed to scale, considering both registration of probes and their execution. While the task of implementing yet another probing mechanism should not be taken lightly, the requirements of a system like Resourceful, which aims for the ability to do pervasive measurements (through hundreds of thousands of probes), have shown that this is necessary.

Looking at current Linux kernels, just the operation of adding `60 000` kprobes to various places will take *approximately half an hour*, while a test server repeatedly crashed when trying to add more than `65 000` probes. This is mainly a design issue of the probing subsystem (adding probes has a $O(n^2)$ complexity).

During probe execution, both the `ftrace` mechanism (mcount) and `kprobes` perform hash lookups in order to find the corresponding probing code. Some types of kprobes actually use the ftrace mechanism for running code at the beginning of a kernel function and will end up doing multiple lookups per probe being fired[9]. Those lookups significantly add to the measured per-probe overheads (see Section 3.3 for an evaluation), and negatively influence instruction caching and pipelining. With numerous kprobes firing constantly, perf reports the probe hash lookup function as the top cycle consumer in the kernel (saturating the CPU). The current design of eBPF (extended Berkeley Packet Filters) also suffers from the same overhead problems, as it only allows attaching eBPF programs to kprobes (with all the disadvantages described above). Other operating systems such as FreeBSD do not have similar issues when registering probes through

[8]The Kamprobes implementation represents a joint effort of myself and Oliver R.A. Chick ; I have implemented the first working prototype in user-space assembly, tail call optimisations and the possibility to avoid calling the `post_handler` when not needed

[9]Masami Hiramatsu has been working on a patch to improve scalability and avoid *multiple* hash lookups, but this is yet to be merged in the mainline kernel. Irrespective of optimisations, the current design involves hash lookups – which kamprobes avoid alltogether.

equivalent mechanisms. However, the design choices made while building kamprobes will also pose interest to them in terms of achieving low-overhead *selective* probing (targeting particular calls to a function and not all of them). This type of probing is important as we aim at tracing very frequent operations, such as firing probes on the software paths in 10 GbE and beyond without dropping packets[10].

For being able to register and fire hundreds of thousands of probes with minimal impact on the system, we have designed an optimised low-level probing mechanism called a kamprobe[11]. This trades off generality and memory footprint for speed of execution. Functionally, a kamprobe behaves like a hybrid between a pure kprobe[12] (which can be placed on any instruction) and a kretprobe (which allows probing function entry and exit points). Instead of targeting function entry points, kamprobes are specialised to probe any *call* instruction. This means they can target particular invocations of a function instead of indiscriminately probing all calls to it (Figure 3.5). Unlike kprobes, kamprobes do not place extra interrupts (`int3`) in the execution path, and will thus also execute faster on platforms where interrupts are slow (i.e virtualized environments where interrupts are forwarded from DOM0).

Similar to a kretprobe, a kamprobe consists of two functions: a `pre_handler` that gets executed before the original (probed) function and has access to the function's arguments and a `post_handler` that gets executed after the original function and will have access to its return value.

For each unique pair of (call site, probing function), the machine code of a custom x86 trampoline is written in an executable memory area. The kernel code at the probed location then gets rewritten on the fly[13] to call into the trampoline instead of the original function. The general structure for such a trampoline is shown in Listing 3.2.

```
; system-call kamprobe on orig_function
probe_rtn:     0x0  ;region to store return address
               ; save return address into region above
               mov %rsp %r11
               mov %r11 #probe_rtn
               ; now save registers as per ABI. this way, the
               ; pre-handler can not interfere with orig_function
               push [r10, r9, r8, rcx, rdx, rsi, rdi, rbx, rax]

               callq [pre_handler]

               ;restore registers
               pop [rax, rbx, rdi, rsi, rdx, rcx, r8, r9, r10]

               ; set return location to #after_orig
               mov #after_orig rsp
               jmp [orig_function]

               ; set stack for return into the original code stream
after_orig:    mov #probe_rtn %r11
               push r11

```

[10]in 10GbE, the minimum interpacket gap is 9.6 ns while for 40GbE that goes down to 2.4 ns; queueing during switching or routing may change the characteristics of data transfers, making them bursty [153]

[11]kernel advanced measurement probe

[12]A kprobe implements the basic low-level kernel mechanism for probing a single instruction. Using this mechanism, the kernel also offers higher-level probes: jprobes (used to probe function entry points and have access to their arguments), and kretprobes (used to wrap function calls by executing extra handlers both on function entry and function exit).

[13]this also takes into account multi-CPU setups where the code being modified might be in-execution on other CPUs

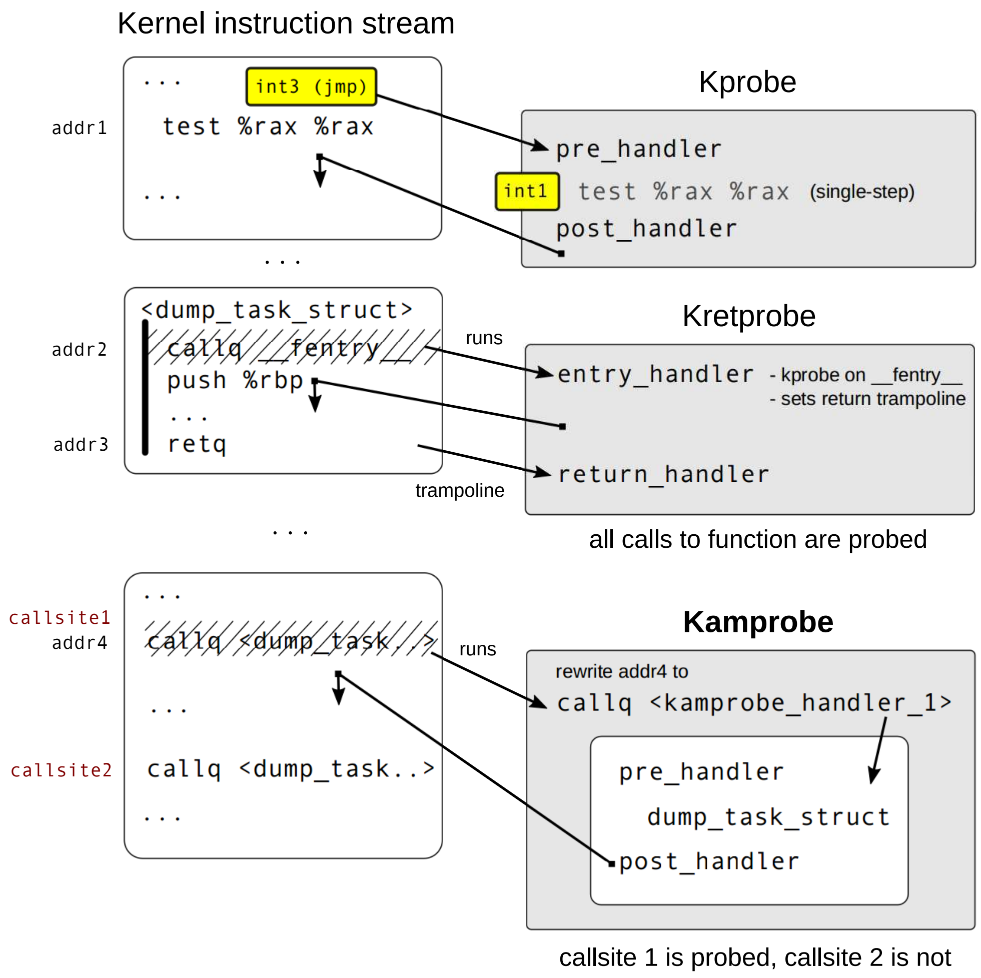


Figure 3.5: Comparison between kamprobes and current Linux probing mechanisms. The hashed areas represent instructions rewritten on-the-fly in the kernel instruction stream. For Kprobes, the `int3` microcode overwrites the first bytes of the probed instruction (`test %rax %rax`). For Kretprobes and Kamprobes, I present a hypothetical probe placed on the `dump_task_struct` function.

```
23              ; post handler returns directly into the original code
24              jmp [post_handler]
```

Listing 3.2: ASM equivalent code for the kamprobe trampolines

Two low-level optimisations have been implemented to further reduce overhead: the first is a tail-call optimisation, set up in lines `20-21`: instead of returning into the kamprobe wrapper, the `post_handler` will return directly into the original instruction stream.

The second optimisation allows for skipping the execution of the `post_handler` when the `pre_handler` returns a non-zero value. Executing the change of the return address previously operated at line 16 in Listing 3.2 is now conditioned on a 0 return value of the `pre_handler`. Taken together, the two optimisations bring the overhead of executing a probe with empty pre and post handlers to that of an additional access to the main memory (evaluation details in Section 3.3).

```
              ...
              callq [pre_handler]

              test %rax, %rax
              jnz #restore

              ;if jnz jumps over those instructions, orig_function
              ;returns directly into the original instruction
              ;stream

              ;0x48 is the number of entries on the stack before
              ;the return value
              mov #after_orig 0x48(%rsp)

              ;restore registers
restore:      pop [rax, rbx, rdi, rsi, rdx, rcx, r8, r9, r10]

              ; set return location to #after_orig
              jmp [orig_function]

              ; set stack for return into the original code stream
after_orig:   mov #probe_rtn %r11
              push r11

              ; post handler returns directly into the original code
              jmp [post_handler]
```

Listing 3.3: kamprobes - optimisation for skipping the `post_handler`

There is a good reason for the current state of low-level probing mechanisms on Linux: they have been designed to be (i) general and (ii) used for targeted instrumentation (tens of probes): this type of instrumentation requires developers to understand the system and to investigate iteratively for the location of the problem. Resourceful proposes a different take, with pervasive instrumentation (tens or hundreds *of thousand* probes) aiming at semi-automatic identification of problematic components/behaviours. As a trade-off, kamprobes are limited to:

- probing at function boundaries
- x86 code (not a limitation of the design but just of the current implementation)
- consuming more memory as the wrapper/trampoline code is duplicated for each probe in order to avoid hash lookups
- no execution safety (unlike an eBPF program, the code executed in the pre and post handlers can easily crash the system or introduce vulnerabilities)

The way they are currently implemented, kamprobes allow the overlaying of other probing mechanisms such as kprobes or mcount calls (the mechanism used by `ftrace`), as long as care is taken to avoid recursion (the kprobes are not probing kamprobe code and kamprobes code is not probing kprobe functions).

Scheduler interposition

Resourceful needs to keep a non-trivial amount of contextual data around, so that each probe knows in what per-process memory region should accounting data be saved, what tokens are currently active, what is the current subsystem nesting level, etc.

In order to access this data with minimum overhead, Resourceful *virtually* augments the kernel `task_struct` data structure. This virtual augmentation is done in order to avoid modifying kernel code directly, which would imply the need to run custom kernels for Resourceful functionality. Instead, we use a set of pid-indexed per-CPU hash tables that are kept up-to-date when processes are scheduled in, out or migrated to different CPUs. Effectively, probes have access to a variable called current_acct (Figure 3.6) which is the equivalent to the kernel variable current (the variable that always points to the task_struct of the current process). Like current, current_acct always points at a data structure containing the current Resourceful context.

Any kernel probes requiring information about whether measurement should be done or not or about what types of aggregation should be configured will access the thread-specific rscfl context through the current_acct pointer. Similarly, the results of any measurement performed for a given pid are written in the corresponding data section. The memory regions themselves can be mapped in user-space by multiple processes, allowing for measurements controlled by other applications (cross-monitoring).

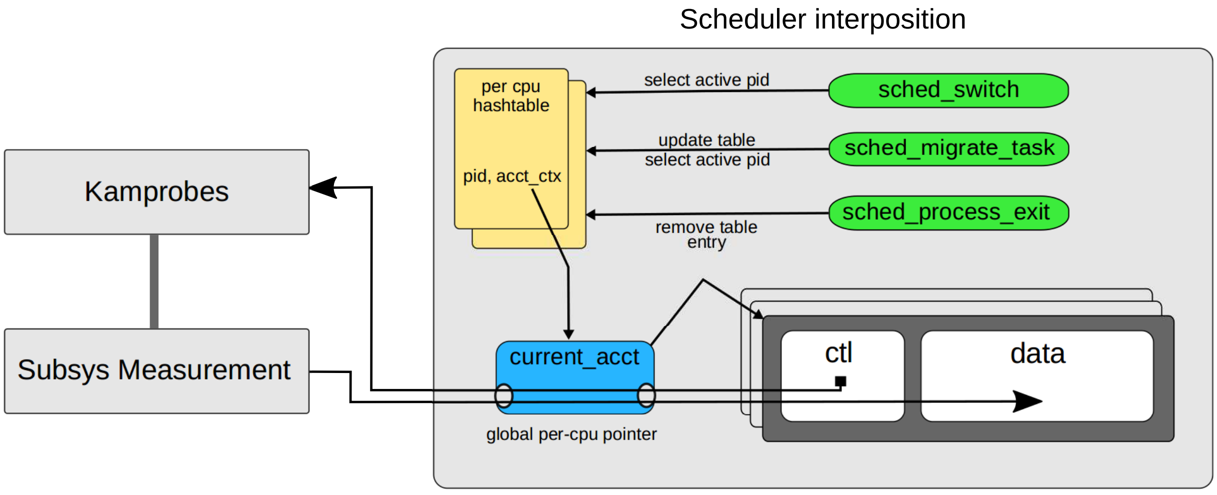


Figure 3.6: Interposing the Linux scheduler

The actual scheduler interposition is implemented using the kernel tracepoints functionality, which allows modules to register functions to be called at particular points in the execution stream. For the interposition, Resourceful uses the sched_switch, sched_migrate_task and sched_process_exit tracepoints.

Kernel measurements

Because of the way the scheduler interposition is implemented (one hash table per cpu) and the way the memory region for writing measurement data is allocated (one region per application thread), we can fire kamprobes and perform measurements without the need of acquiring locks. This reduces per-probe overheads and simplifies the design of the system.

Furthermore, the API exposed to user space allows for zero-copy access to the measured data. This assumes that aggregation is configured to be done kernel-side and is limited to measurements of synchronous execution (the execution of the probed kernel paths and of application code does not overlap). Using this feature means one might need to increase the size of the kernel-level per-thread Resourceful data region: using zero-copy means some data elements will not be freed as soon as possible, which can lead to buffer overwrites on the kernel side and loss of measurement data (the user space application is informed when this happens). Alternatively, the data can be copied to user-space whenever the activity associated with a given token ends.

3.2.3 User space API

The resourceful user-space API allows the applications to:

1. configure kernel-side aggregation and the set of active kamprobes

2. express interest in the resources consumed by particular system calls (one-shot), or start/stop measurements while associating them with a given activity token;

3. define activities based on meaningful application semantics (i.e. “processing a request”) – resource consumption reports are then be aggregated per activity, in addition to their per subsystem aggregation.

4. read the aggregated data and run code based on the measured values

The main role of the API is to abstract away from applications the details required for setting up and reading kernel-side measurements. Because of this, the application is not exposed to any of Resourceful’s implementation details, and in addition high level behaviour can be implemented on top of the basic operations. The underlying system allows for multiple processes to use Resourceful simultaneously: each application thread receives a separate reserved memory buffer when initialising (call to `rscfl_init`). Applications are able to express interest, define activities and measure resource consumption independently from one another. However, the current implementation considers the kernel subsystems where measurements are performed to be defined at the time when the kernel module is loaded (i.e it’s not possible for one application to measure just time spent in the TCP stack while another tracks time spent in VFS – both applications will receive data from both subsystems).

Listing 3.4 briefly shows some of the important data structures used by the API. On line 15, `rscfl_handle` can be seen as an opaque structure by the application, with each thread getting its own such handle. The internal data fields of the handle are used by the API to access the two memory regions shared between the kernel and user space: the data region (the buffer at line 17 - with the underlying structure being `rscfl_acct_layout_t`) and the control region (the member at line 18).

The control memory layout is used to make sure that (i) there is version compatibility between the version of the API library and the loaded Resourceful kernel module; (ii) the API has access to tokens and (iii) it can express interest in starting the measurements or stopping them.

On the other side, the data layout is split into two regions: an array of `accounting` structures that represent each aggregated measurement and contain an index of all kernel subsystems that were touched. The second region contains the actual data for those subsystems (with the various PMU and kernel counter values). This arrangement allows a fixed size for each of the regions while not requiring each aggregated measurement to touch the same number of subsystems.

```
1  struct rscfl_acct_layout_t
2  {
3    struct accounting acct[STRUCT_ACCT_NUM];
4    struct subsys_accounting subsyses[ACCT_SUBSYS_NUM];
5  };
6
7  struct rscfl_ctrl_layout_t
8  {
9    unsigned int version;
10   syscall_interest_t interest;
11   int new_tokens[NUM_READY_TOKENS];
12   int num_new_tokens;
13 };
14
15 struct rscfl_handle
16 {
17   char *buf; // opaque field, data organized as rscfl_acct_layout_t;
18   rscfl_ctrl_layout_t *ctrl;
19   int ctrl_fd; // for ioctls
}
```
Listing 3.4: Important Resourceful data structures

The operations available to userspace are shown in Listing 3.5, grouped by purpose. As can be seen, a small number of functions is sufficient for making use of all Resourceful functionality: the application needs to first obtain a handle by calling `rscfl_init()`, can retrieve opaque tokens representing each high-level activity (`rscfl_get_token`), can start/stop accounting and then read the results. This is a two-stage process: first retrieving an index into the results associated with every activity (`rscfl_read_acct`) and then accessing the actual subsystems through `rscfl_get_subsys`.

The default aggregation of resource consumption metrics happens at the level of an activity token, and is done in the kernel: all resources consumed by a user token are reported in a single data structure. Besides doing aggregation at the level of the Linux kernel, it is possible to aggregate things in user-space as well (this is usually needed for asynchronous operations of type 1, when the application finds out in what data structure the measurements should be aggregated into *after* the kernel has already done the measurements). The `rscfl_merge_acct_into` function is used for this purpose. Listing 3.6 presents a simple example of using activity tokens to record system call metrics. All aggregation happens in-kernel.

```
// initialisation
rscfl_handle rscfl_init();
rscfl_handle rscfl_get_handle(void);

// tokens (tracing application activities)
int rscfl_get_token(rscfl_handle rhdl, rscfl_token_t **token);
void rscfl_switch_token(rscfl_handle rhdl, rscfl_token_t *token_to);
int rscfl_free_token(rscfl_handle rhdl, rscfl_token_t *token);

// accounting
int rscfl_acct(rscfl_handle rhdl, rscfl_token_t *token,
               interest_flags fl);
int rscfl_read_acct(rscfl_handle rhdl, accounting *acct,
                    rscfl_token_t *token);
subsys_idx_set* rscfl_get_subsys(rscfl_handle rhdl,
                                 accounting *acct);

// user-space aggregation
subsys_idx_set* rscfl_get_new_aggregator(unsigned short no_subsys);
int rscfl_merge_acct_into(rscfl_handle rhdl, accounting* acct_from,
                          subsys_idx_set *aggregator_into);
```
Listing 3.5: The basic Resourceful API

The API also provides high-level functionality based on the operations described so far. In particular, I have implemented the ability of doing map-reduce[14] style processing of results in user-space: for example, the application would be able to select a given field inside measurements (for example, CPU cycles or memory consumed) and provide a reduce function that operates on the field across multiple subsystems. This can be used for example for easily obtaining sums (total number of cycles across all kernel subsystems) or for doing other mathematical processing (min/max/standard deviation). However, if data in each subsystem is of interest (as is the case

[14]meaning from functional programming, considering map/filter/fold operations

in evaluations performed in this thesis), the per-token aggregation happening in the kernel is sufficient.

```c
//...assume rhld is an initialised Resourceful handle
FILE *f1, *f2;
rscfl_token_t token1, token2;
accounting acct1, acct2;

rscfl_get_token(rhdl, &token1);
rscfl_get_token(rhdl, &token2);

rscfl_acct(rhdl, &token1, ACCT_START);
f1 = fopen("data.md", r);
fprintf(f, "writing to file 1\n");
//...
rscfl_switch_token(rhdl, &token2); // token1 becomes inactive
//...
f2 = fopen("data2.md", w);
fprintf(f, "writing to file 2\n");
fprintf(f, "again...\n");
rscfl_acct(rhdl, NULL, ACCT_STOP);

//read resources consumed by token1 (2 system calls)
rscfl_read_acct(rhdl, &acct1, &token1);

//read resources consumed by token2 (3 system calls)
rscfl_read_acct(rhdl, &acct2, &token2);
```

Listing 3.6: Example using activity tokens

3.3 System evaluation

3.3.1 Evaluation goals

The evaluation presented in this chapter focuses on evaluating Resourceful's ability to fulfil stated goals. To that end, a series of microbenchmarks will show that the low-level probing mechanism (kamprobes) introduces smaller overheads than existing probing methods, both for system calls doing very little actual work as well as for ones where the baseline latency is higher. This characterisation will show worst-case behaviour as in practice the cost of slightly more expensive system calls will be amortised by application operations.

Then, I will switch focus to a series of macrobenchmarks, testing both the performance perturbation that Resourceful induces on two real applications under load (`lighttpd` and `fio`) as well as the sanity of resulting measurement data. Here, the goal is showing that Resourceful induces minimal shifts in the latency distributions of application operations and does not produce statistically significant decreases in throughput. This is essential as it suggests that when measuring Resourceful is unlikely to become itself the bottleneck.

There is a separate discussion on showing that the underlying premise of Resourceful is reasonable: I will show that tracking asynchronous resource consumption is important in explaining latency variations; Furthermore, I will pick a workload where the bottleneck is not immediately obvious (i.e the workload is not simply I/O-bound or CPU-bound) and determine how much of the latency can be explained by measurements taken using Resourceful.

Lastly, I will set to prove that using the same data to answer simple what-if questions is *possible* under certain constraints. Relaxing those constraints is the topic of Chapter 4.

Methodology

The results described in the evaluation sections of this thesis are based on real system measurements done using Resourceful or other tracing and probing tools, for comparison. Most of those measurements come from inspecting complex systems that can be controlled through numerous configuration parameters, and are affected by noise. The realities of such system measurements place the analysis of results in a non-standard context:

1. typical distributions are strongly non-normal, hard to parameterise and often multi-modal[15]: the mean is a poor statistic for describing them, and we are often interested in the behaviour of the system in the tail region of the distribution; Multi-modality is to be expected given the existence of deep memory hierarchies and due to the presence of various caches in different layers (application/kernel). Each caching level may generate it's own peak in the observed distributions.

2. perceived non-normality may exist as a consequence of underlying population groups and factors that are unknown at the start of the analysis. In most cases, deciding on a data stratification strategy before conducting the measurements is challenging. Indeed, part of the analysis process described when evaluating Resourceful results is focused on isolating such groups;

3. relationships between different random variables are sometimes non-linear and sometimes even non-functional (a simple example of such a non-functional relationship would be an "X" shape or multiple parallel lines within a scatter plot).

As a direct consequence of point 1, the statistic evaluation will be done directly on non-parametric, empirical distribution functions (derived directly from the measurement sample) instead of trying to fit known distributions for estimating the underlying populations' characteristics. Most of the time, we will be simply using the plug-in principle to estimate population distributions, considering a large number of samples. In order to understand how confident we can be in estimations of particular statistics (median, standard deviation, percentiles) under such conditions, we will use bootstrapping methods for deriving confidence intervals. When hypothesis testing is needed for determining whether two samples come from the same population or not, we will similarly try to apply non-parametric tests such as Mann-Whitney's U test instead of the standard t-test (which works best in the context of normal distributions).

Because of point 3, measures based on the mean, like correlations, are not always appropriate in describing the strength of relationship between two variables. Even though they are harder to estimate without bias, information theory entropy-based measures such as mutual information will be used to perform variable selection.

Overhead measurement We normally look at overheads when comparing the behaviour of a system in terms of a given parameter (e.g. latency, throughput) between two independent experiments. For example, we would like to analyse the overhead imposed by an application running with Resourceful probes and doing measurements when compared to the application running without any probing. Each experiment is seen as a random variable (X_{base}, X_{rscfl}), with the corresponding empirical distributions built from the measured samples. I will contrast the usual approach to the methodology typically considered in the evaluation of the thesis:

Normally, the analysis target is to determine a confidence interval for the difference between the means of the two samples $\bar{x}_{rscfl} - \bar{x}_{base}$, at a given significance level (.05)[16].

[15]this is the reason one typically investigates "tail latency": a non-trivial percentage of samples fall towards the right-end of the distribution; normal distributions are not characterised by "long tails".

[16]A 0.05 significance level tells us that when repeatedly computing the confidence intervals from new samples of the

The main issue of this approach, besides the well known difficulty in interpreting p-values and confidence intervals intuitively, is that we're not necessarily interested in how the *mean* of the overhead behaves. One might rightfully be interested in a worse-case overhead (99.9 percentile) that remains acceptable. Can I run Resourceful on 500 leaf services that are used for answering a single request? (as previously described in Note 3.2)

Answering those questions requires the identification of a distribution of possible overheads given the current samples from X_{base} and X_{rscfl}. To compute this distribution, we introduce a new random variable, $X_{ovhd} = X_{rscfl} - X_{base}$ to represent the difference between the two previous random variables. The probability distribution function of X_{ovhd} can be defined in terms of the joint probability of the original two variables:

$$P(X_{ovhd} = k) = \sum_{i=-\infty}^{\infty} P(X_{rscfl} = i, X_{base} = j), \text{ with } i = j + k$$

which becomes a convolution if we consider X_{base} and X_{rscfl} independent:

$$P(X_{ovhd} = k) = \sum_{i=-\infty}^{\infty} P(X_{rscfl} = i) \cdot P(X_{base} = i - k)$$

In practice, we will use a FFT convolution to efficiently estimate the result numerically given a large number of samples. This gives us the complete *distribution* of overheads, from which we can pick particular statistics of interest (e.g. the 99th percentile) and compute corresponding confidence intervals. The only less intuitive aspect is the fact that if the distributions of X_{rscfl} and X_{base} overlap, then the distribution of X_{ovhd} will also contain negative values. This needs to be understood in the sense that there are samples in the X_{base} distribution which are higher than samples in the X_{rscfl} distribution.

Experiment design

The expected overhead of firing one empty kamprobe is below 100 cycles / 30 ns. Therefore, accurately measuring those overheads poses a challenge: they will be in the order of an extra access to the main memory. To have any confidence in such overhead measurements, the initial set of experiments (microbenchmarks) will be executed in a tightly controlled environment:

1. `isol_cpus/taskset`: one of the CPUs is isolated from the kernel SMP balancing and scheduling algorithms. This assures that, excepting the process doing the benchmarking, no other processes is being executed on that core. Furthermore, the benchmarking process is not interrupted by the scheduler.

2. IRQ tuning: in some of the experiments (this will be mentioned for every experiment using the technique) I have diverted interrupts away from the CPU doing the measurements. This is done in order to eliminate possible variation due to interrupt service routines executing arbitrarily on CPUs doing Resourceful measurements. However, it is not always realistic to impose such tuning, as real systems will most likely either not tune IRQs or use different settings than the ones picked in this setup

3. TCP tuning: a number of TCP stack options have been set in order to account for the approximately loss-free network in which we conduct experiments, as well as the need for lots of connections per second (port reuse is particularly important)

same population (in our case, the confidence interval for the difference in mean values between two configurations), the interval will contain the true difference 95% of the time. In other words, there is a 5% chance that the determined interval does not contain the true overhead.

4. cache warming: a number of the same operations as the ones being measured are performed before starting the actual measurement, in order to warm up caches (from libc, kernel or the CPU).

5. batched measurements: instead of measuring a single extremely short-lived operation, we measure 1000 of them and then obtain the average per operation. This improves the accuracy of results but might also filter out some outliers or really slow operations from the data. We can balance the size of batching if we observe too little variation in the final results.

6. timers: we perform raw TSC measurements (cycles), and obtain the wall clock time using `CLOCK_MONOTONIC`[17]

All measurements are done on a server with an 8 core Intel Xeon E3-1231 v3 CPU at 3.40GHz (1 ns = 3.4 cycles). A constant TSC is available for this CPU, and reading the hardware cycle counter has also been enabled for virtual machines in experiments that use them. The server has 32Gb of RAM. Processor frequency scaling and other *software* throttling mechanisms have been disabled. The Linux kernel version used in the experiments is 3.19.

Kamprobe microbenchmarks against other probing mechanisms

The evaluation of `kamprobes` as they are used in Resourceful starts through a series of microbenchmarks that highlight the behaviour of the measurement framework in a worst-case scenario: performing lots of system calls in a tight loop. In practice, the expected overheads will be amortised over the system call and a number of operations performed by the application in user space. However, applications executing lots of system calls could expect overheads like the ones shown below.

I focus on three system calls: `socket`, `read` and `write`. The `socket` system call has been chosen as it does little work and is normally very fast: it mainly allocates resources in the network stack and then creates a corresponding file descriptor that can be handed to user space applications. The `read` and `write` calls show the behaviour of probing on the corresponding paths in the kernel, dependent on the size of reads and writes, kernel caching, etc. Those are system calls that can block and are more representative of calls that do a non-trivial amount of work in the kernel.

For each system call, we study the distribution of the number of cycles[18] until return, in the following situations:

baseline No kernel-side tracing or measurements enabled. This is close to the latency of the system call on our hardware under normal circumstances.

rscfl-idle The Resourceful kernel module is inserted but performs no active measurements. The overheads here are simply caused by the probes checking that no accounting should take place and returning. This is close to the minimal overhead that can be imposed by `kamprobes`, both when considering monitored applications and other applications running on the same system.

rscfl This shows the overhead of firing all required `kamprobes` (on crossing subsystem boundaries) and doing the actual Resourceful measurements for the given system call.

[17]`using CLOCK_MONOTONIC_RAW` sometimes imposes an additional unwanted latency; for the durations of time being measured I do not expect significant impact due to NTP or `adjtime` incremental time adjustments.

[18]cycles were chosen as a measure instead of other possibilities such as wall clock time because they are inherently cheap to measure (by reading the TSC), and thus will influence the experiments as little as possible. For measurement, I use the methodology recommended by Intel in one of their whitepapers [109], in order to limit the effects of instruction reordering and inherent measurement variance. In particular, I use the `cpuid` and `rdtscp` instructions.

rscfl-ftrace In the way it was defined above, the rscfl series is not directly comparable to ftrace, because it collects much more data on crossing from one subsystem to another. For this reason, I have modified Resourceful for enabling a closer apples-to-apples comparison with ftrace. In this mode, Resourceful only snapshots the cycle counter on every function entry. Ftrace also does a slow symbol lookup for the function while we only record its address (but symbol lookup can easily be executed asynchronously). However, ftrace caches the symbol resolution, and that will happen during the experiment cache warmup phase (unless the number of functions touched during the experiment exceeds the cache size). I therefore claim that rscfl-ftrace is *functionally* equivalent to ftrace using the function tracer.

kprobes This is equivalent with the rscfl series, but uses the normal Linux kernel probing mechanism (kprobes) for performing the required Resourceful measurements. It provides for a direct evaluation of the advantages of `kamprobes` when relating to existing mechanisms.

ftrace enables ftrace with the function tracer while running the microbenchmark. Other tracers are expected to be even more expensive than the results shown here.

Those measurements are split into two groups in order to avoid cluttering the same graph: the first looks at a comparison with `ftrace`, while the second shows the rest of data, highlighting the behaviour under various Resourceful configurations.

The subsequent figures detailing the actual overhead distributions (3.8, 3.9) are computed as described at the beginning of this section. In the title of those graphs, I first mention the baseline and then the test random variable; The shown overheads represent a distribution of the test – baseline random variable.

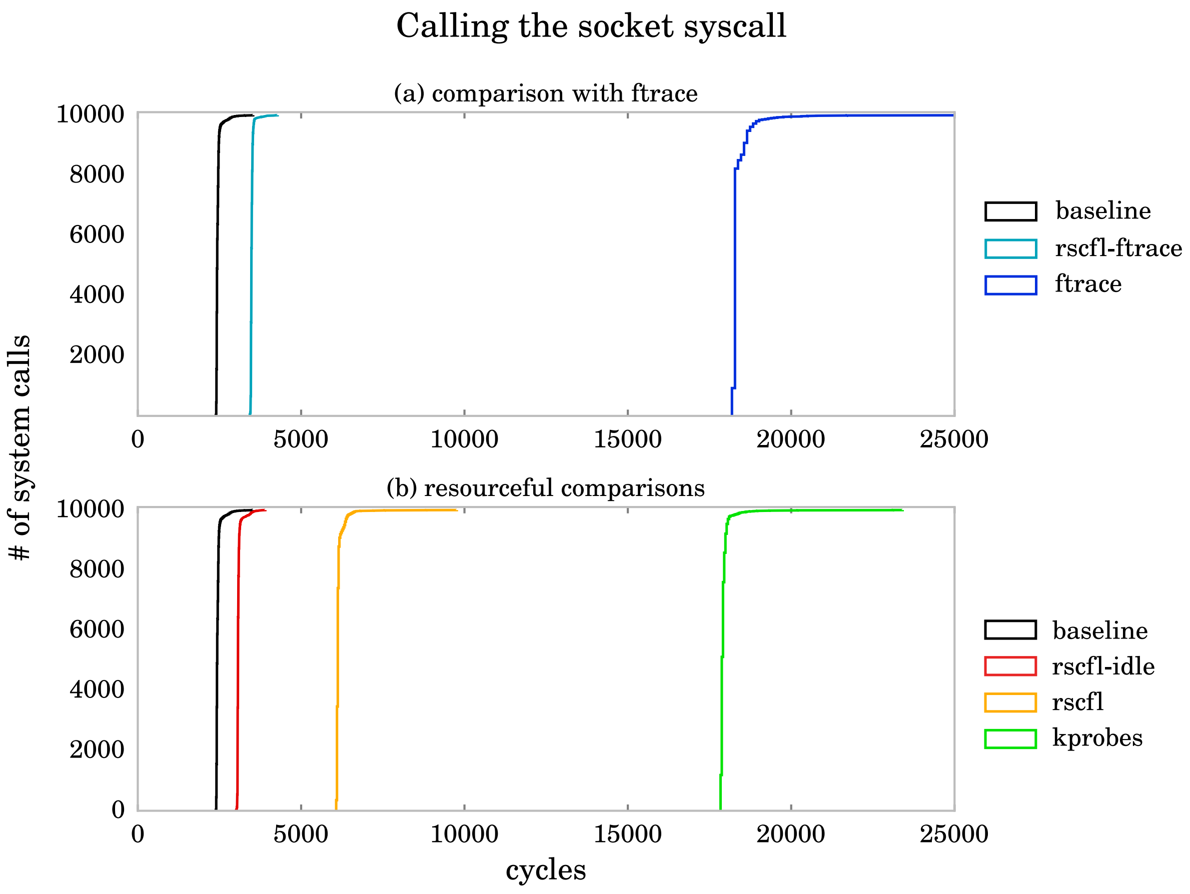


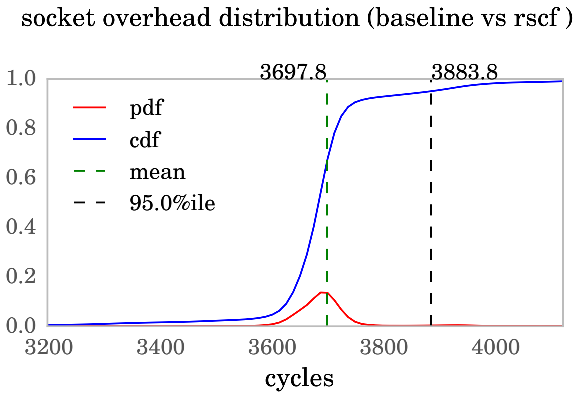
Figure 3.7: Cumulative histograms representing cycles required for calling a socket syscall under different conditions. Plot (a) shows a comparison between using `kamprobes` to implement ftrace functionality (`rscfl-ftrace`) and typical `ftrace` overheads when using the function tracer. Plot (b) looks at Resourceful overheads, when the kernel module is inserted but not performing measurements (`rscfl-idle`), doing measurements based on `kamprobes` (`rscfl`) and performing the same measurements using existing kernel probing mechanisms (`kprobes`).

Socket system call: discussion The first thing to notice in Figure 3.7 is that all probing methods based on hash lookups, like `ftrace` or `kprobes` are significantly slower than `kamprobes`. `Ftrace`, besides being the slowest in the group, also introduces more variability in the time it takes to execute system calls when compared to other methods. Figure 3.8b further confirms this observation, showing that just the overhead of using `ftrace` instead of `rscfl-ftrace` (based on kamprobes) is almost 6x the time it would take to execute the socket system call without any probes[19]. A further major issue with `ftrace` is its selectivity: any filter that an end-user adds (for example, for limiting probing to a subset of kernel functions, like required by Resourceful) introduces *additional* overhead[20].

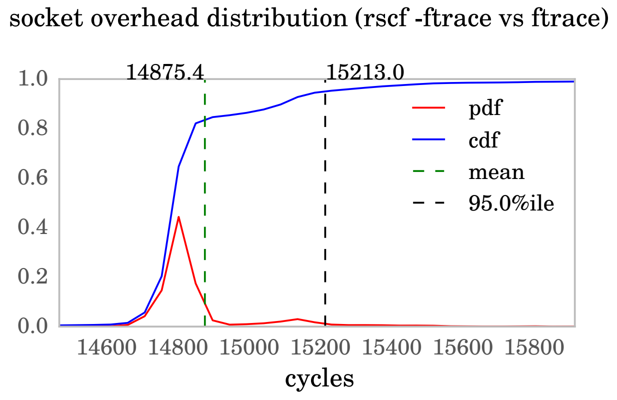
In terms of Resourceful measurements based on the `kamprobe` mechanism, Figure 3.8a shows that they introduce a 1.47x overhead relative to the baseline. This might seem high when expressed as a percentage, but one needs to remember that the socket call is extremely fast in the first place. The result also needs to be put into perspective by considering the alternative probing mechanisms, as we've seen for `ftrace` (6x). Another comparative data point is obtained by switching the Resourceful probing method to `kprobes` but keeping the measurements they exe-

[19]Mean overhead (14875.4) divided by mean baseline (2500) = 5.95

[20]No filters were added in the current experiment

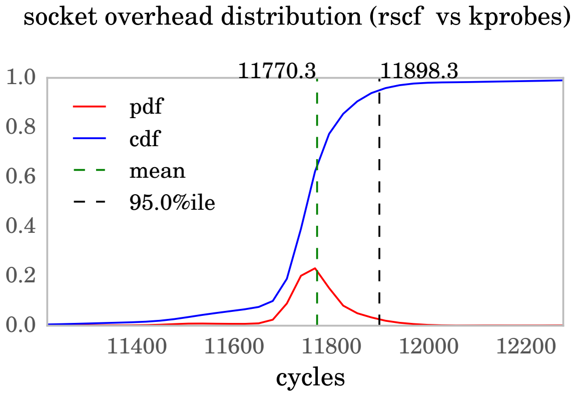


(**a**) Overheads imposed by accounting with Resourceful

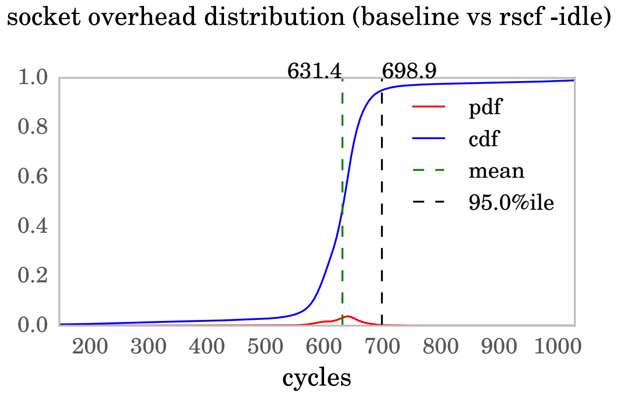


(**b**) Overheads of ftrace compared to rscfl-ftrace

Figure 3.8: Overhead distributions showing the expected impact of Resourceful using kamprobes and the (comparably) significant overheads imposed by running ftrace



(**a**) Comparing probing methods (kprobes - kamprobes)



(**b**) Probing effects of kamprobes

Figure 3.9: Effects of probing

cute the same. This results in the significant overheads shown in Figure 3.9a (4.7x on average when compared to the baseline).

Probing effects: Because the socket system call does very little actual work before returning, this microbenchmark also offers the opportunity of understanding the probe effects that one might expect from kamprobes with respect to time overheads. Considering that a socket system call crosses 16 subsystem boundaries and that a `kamprobe` is fired on each crossing, we summarise the results in Table 3.1. The `rscfl-idle` configuration represents overheads that would be incurred by applications *not* performing Resourceful measurements but executing probed codepaths. The `rscfl` configuration considers non-empty probes running actual Resourceful measurements. Probe effects also include any type of bias introduced in the measured data. A thorough evaluation of those effects is left as future work.

Configuration	Overhead (cycles)		Overhead (ns)[21]		Computed from
	avg	95%ile	avg	95%ile	
rscfl-idle (probe effect)	39.46	43.68	11.6	12.8	Fig 3.9b
rscfl	231	242.7	67.9	71.3	Fig 3.8a

Table 3.1: Summary of Resourceful probe effects

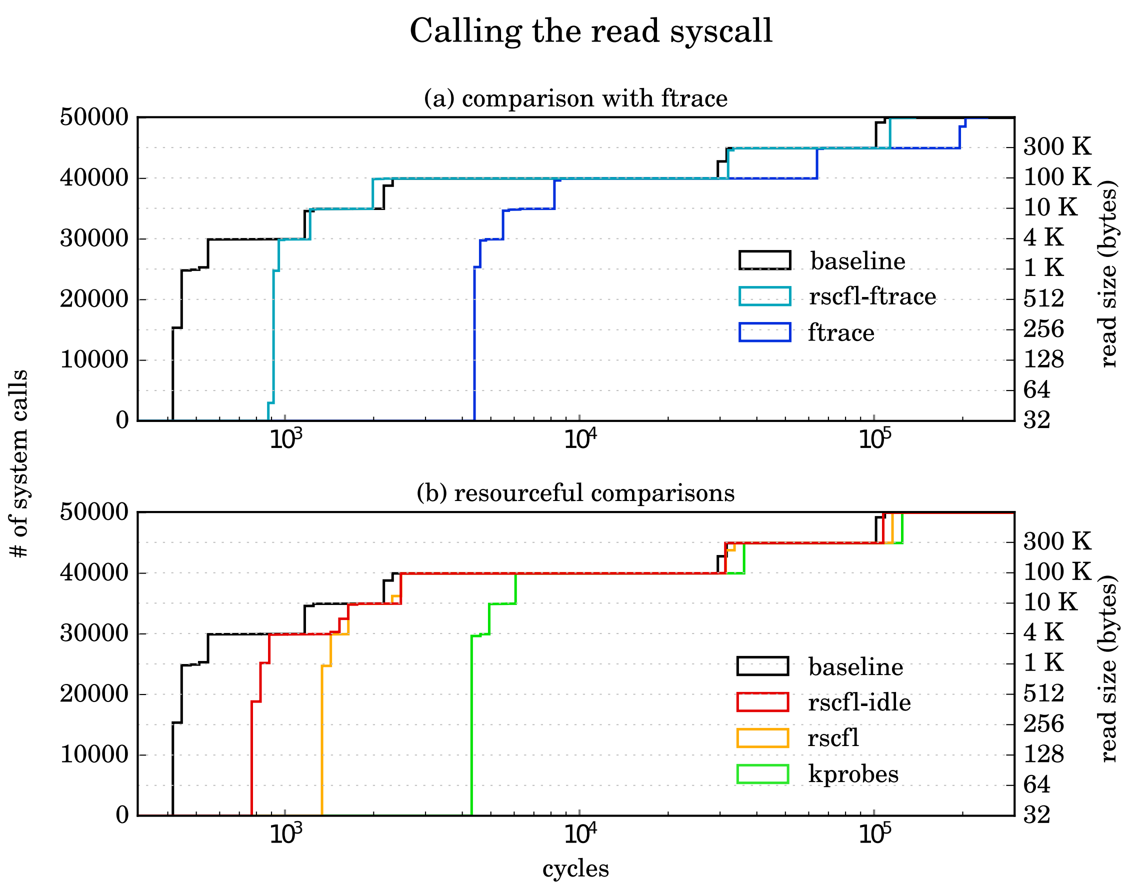


Figure 3.10: Cumulative histograms representing cycles required for calling a `read` syscall under different conditions. The meaning to the series remains as before; Please note the logarithmic x axis. The shape of the cumulative histogram is atypical because we have represented the cycles required for fulfilling reads of different sizes on the same graph. The sizes are shown on the secondary y axis marking the start of samples with reads of that size. For each read size, 5000 samples were collected.

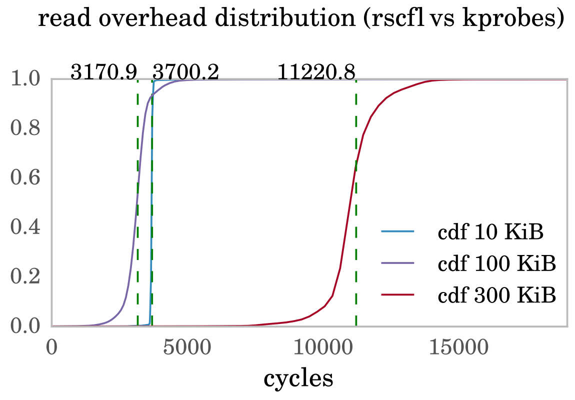
Read system call: discussion In order to avoid plotting a 3D histogram which is harder to interpret, Figure 3.10 presents the cumulative histogram for the system call without stratifying on the read size. This is problematic only for the small read sizes, where the distributions overlap. We can identify each of the larger read sizes represented by staircase jumps towards the same y value. Identifying the x value for which each of the curves reaches a given y can then be used to compare like-for-like probing results for read calls of the same size.

We can still clearly observe the relative differences between the experiments: the ranking of different probing or tracing methods stays identical to the one discussed for the socket system call. Because of the logarithmic x axis, constant overheads are represented as smaller and smaller intervals moving towards the right, as can be seen in Figure 3.10a.

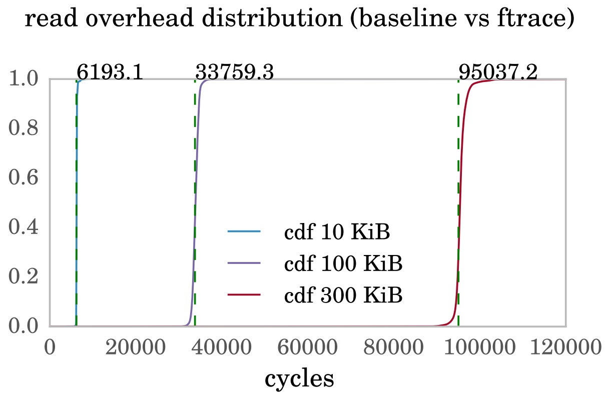
In terms of a detailed look at the overheads, we will compare the Resourceful measurements under two probing methods (existing – `kprobes` and the one proposed in this chapter – `kamprobes`). Secondly, we will take a look at how `ftrace` performs in relation to the baseline to give a sense of just how much slower is this tracing method when compared to Resourceful measurements.

For this particular experiment, the Resourceful measurements using `kprobes` scale well, although the absolute overheads remain worse than what `kamprobes` provide, across all read sizes

[21]Considering a fixed CPU frequency of 3.4 GHz



(a) Kprobes overhead when compared to kamprobes for medium-sized reads



(b) Ftrace overheads for medium-sized reads

Figure 3.11: Overheads for the `read` system call. The mean overhead is marked by a dashed line. The number after 'cdf' in the legend of each graph represents the read size in bytes.

(as shown in Figure 3.11a). Here, the overhead stays approximately constant for reads between 10 KiB and 100 KiB bytes, but then starts increasing linearly.

In contrast, the `ftrace` overheads for large read sizes shown in Figure 3.11b are significantly higher, and increasing linearly with the read size.

The overhead of Resourceful probing reads using `kamprobes` varies between 8% (for large reads) and 20% (for small reads) when compared to the baseline. Using the same reference, the minimum `ftrace` overhead is 2.5x

Write system call: discussion Figure 3.12 follows the same conventions described for the read system call. However, here we see both `kprobes` and `ftrace` at a big disadvantage when compared to the proposed probing method: the differences in both (a) and (b) increase as the write size increases. `Ftrace` performs particularly poorly in this microbenchmark.

Even without going into detailed experiments for diagnosing ftrace slowdowns, a number of possible explanations exist, related to the fact that the write system call touches numerous functions in the kernel (as the writing is performed in chunks), presenting a number of challenges for `ftrace`:

- the number of symbols that `ftrace` searches for might exceed the size of the symbol cache, leading to expensive continuous evictions and updates
- the execution of numerous functions might lead to exhausting the tracing buffer, leading to significant slowdowns. The fact that resourceful is able to perform in-kernel aggregations makes it immune to this issue.
- latencies also compound as the (larger) `ftrace` overhead is repeatedly incurred on functions called in a loop. This can also be observed for the evolution of the difference between `rscfl` and `rscfl-trace` in (a), but on a much smaller scale.

The overhead of Resourceful probing writes using `kamprobes` varies between 20% (for large writes) and 44% (for small writes) when compared to the baseline. Using the same reference, the minimum `ftrace` overhead is 7.7x

Microbenchmark conclusions As described initially, the overheads mentioned in this section should be seen as worst-case scenarios, as we are exercising a particularly taxing use case for tracing and probing mechanisms: an application (the benchmark) that does nothing else but

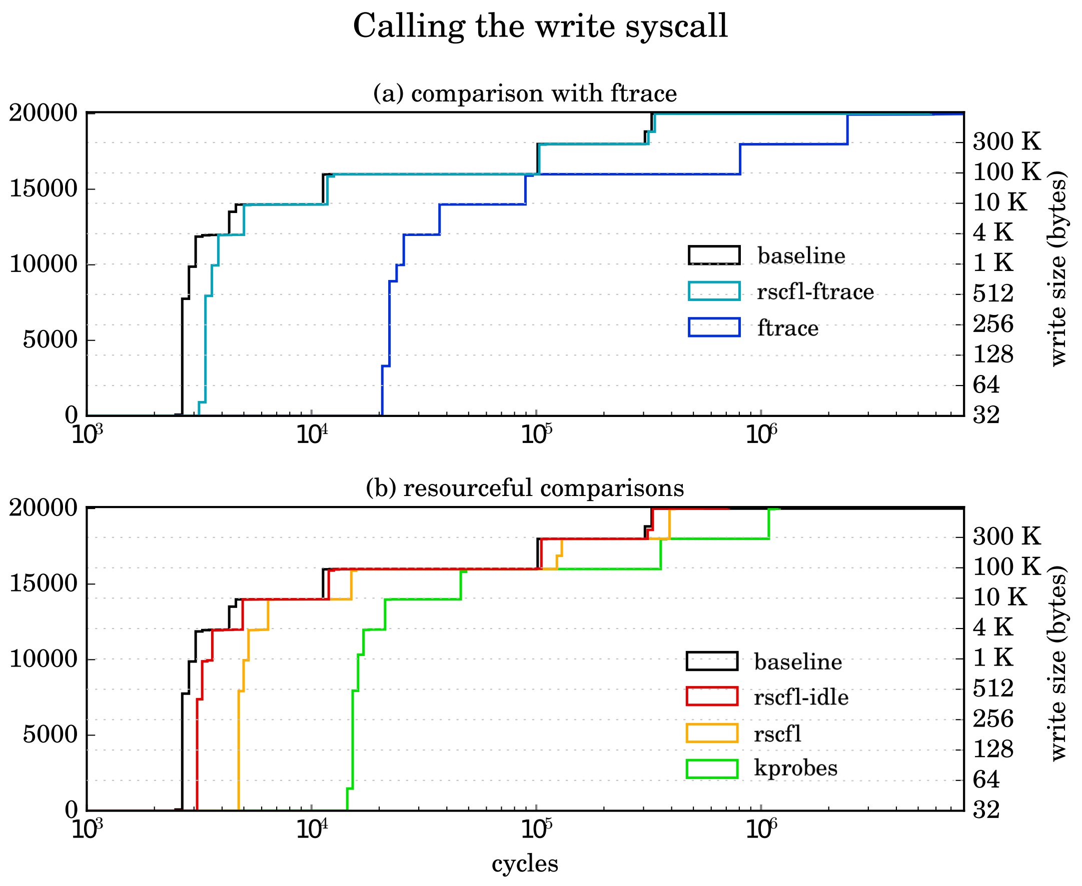
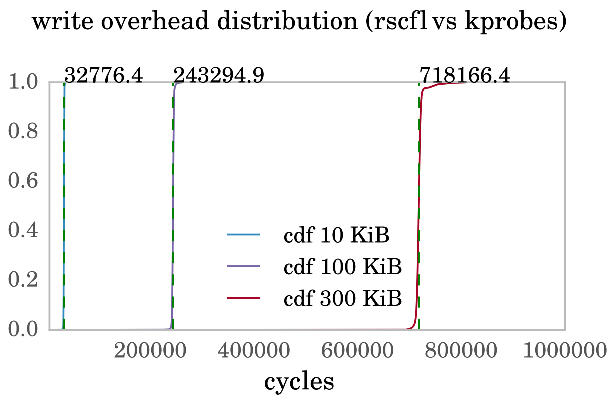
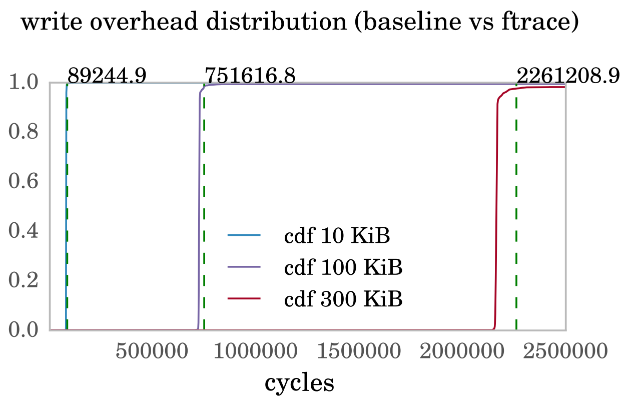


Figure 3.12: Cumulative histograms representing cycles required for calling a `write` syscall under different conditions. The meaning to the series remains as before; Please note the logarithmic x axis. The shape of the cumulative histogram is atypical because we have represented the cycles required for fulfilling writes of different sizes on the same graph. The sizes are shown on the secondary y axis marking the start of samples with writes of that size. For each write size, 2000 samples were collected.



(a) Kprobes overhead when compared to kamprobes for medium-sized writes (more than 1 page)



(b) Ftrace overheads for medium-sized writes (more than 1 page)

Figure 3.13: Overheads for the `write` system call. The mean overhead is marked by a dashed line. The number after 'cdf' in the legend of each graph represents the write size in bytes.

execute system calls in a tight loop. Even so, I have shown that kamprobes behave significantly better when compared to existing probes such as kprobes. Furthermore, I have proved that

`fio` workload	Latency increase (%)			IOPS reduction (%)	
engine, threads, r/w mix, block size	min	50%ile	95%ile	min IOPS	max IOPS
psync, 1, rand w, bs=32K	0.02	6.12	5.32	19.1	4.19
psync, 4, rand w, bs=32K	0.1	8.79	1.04	18.2	3.08
psync, 4, rand r70%w30%, bs=32K	2.27	13.5	8.97	9.44	0.1

Table 3.2: Slowdown imposed by Resourceful when running the `fio` benchmarking tool. All workloads are bypassing the buffer cache by using `DIRECT_IO`

implementing tracing and measurement tools on top of `kamprobes` yields significantly smaller overheads when compared to `ftrace`.

With worst case overheads between 1.47x for very fast system calls and under 20% for system calls performing a non-trivial amount of operations, `kamprobes` may be considered as probing mechanisms in production systems, especially as alternatives (like ftrace or kprobes) are *always* slower. To get a better idea of real-world overheads, I will next look at instrumenting existing applications.

Real application behaviour

Having analysed the probing method (`kamprobes`) in microbenchmarks, I now turn to examining the overheads imposed in more realistic situations. The focus also moves from individual probes to the overhead imposed by the Resourceful framework as a whole. For this purpose, I have modified a popular I/O benchmarking tool, `fio` and a web server, `lighttpd` to use the API described in this chapter and record the resources consumed for performing individual I/O operations or responding to http requests.

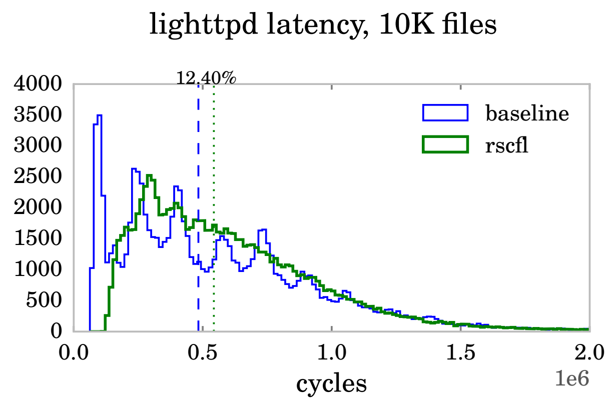
The overheads imposed by Resourceful are not tightly connected to whether a particular application workload is CPU-, network- or I/O-bound. What matters instead is the number of system calls that the application makes every second, as well as the complexity of the code that runs on the kernel-side given the system call (as it could cross kernel subsystems and thus trigger the execution of numerous probes). As a consequence, I have picked workloads which would make heavy use of system calls (writing small block sizes for I/O workloads and serving as many small files as possible in the case of lighttpd).

For `fio`, we expect the numbers to describe results close to the worst-case scenario (as benchmarking tools are intentionally pushing the system to its limits). Three workloads using the default I/O engine are measured, using either 1 or 4 concurrent submission threads. In all cases, the workload was I/O-bound, and more importantly, required numerous system calls: each thread writes 1Gb to disk using a 32K block size. It is not meaningful to compare the overheads across two workloads as the read/write patterns were random.

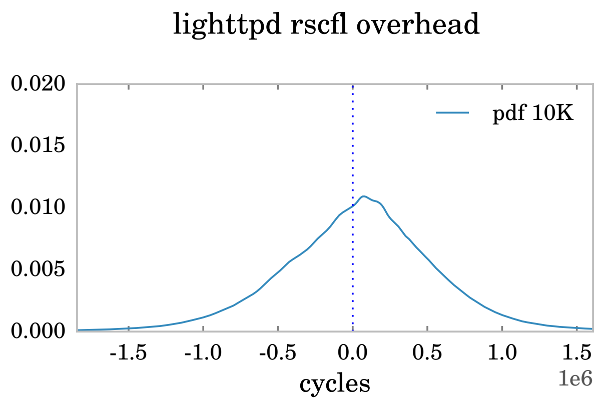
I summarise the differences between running each workload with and without Resourceful in Table 3.2. The increase in latency at given percentiles has been measured, together with the reduction in both the minimum and maximum number of IOPS that were achieved. In each case, there is a small shift in the minimum latency achievable and a more important (max 13.5%) perturbation of the median. However, Resourceful does not add significant further queuing along the I/O path, as shown by the limited perturbation of the 95^{th} percentile. Reductions in throughput (IOPS) are, as expected, largest during periods of contention (when the achieved number of IOPS was smallest).

For evaluating `lighttpd`, I compare server-side request latencies[22] taken when the server runs normally (this is the baseline) with server-side latencies taken with Resourceful active and per-

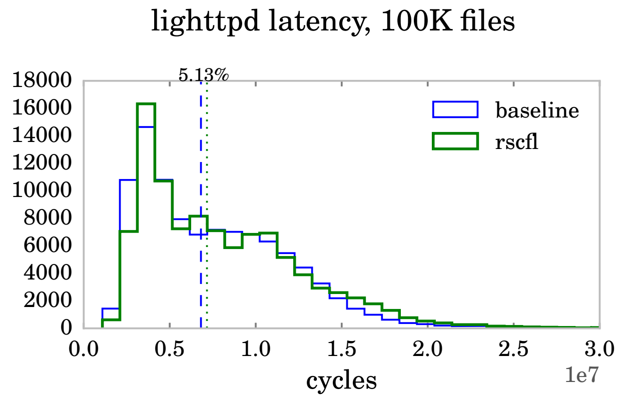
[22]Side effects of Resourceful to end-to-end latency that are indirect (for example, because of timing changes that could affect the TCP congestion control algorithms) have not yet been analysed.



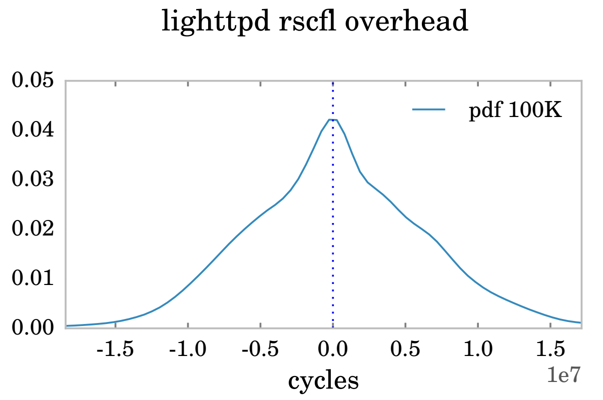
(**a**) Change in lighttpd latency histogram, 10K files. The shift in median expressed as a percentage of baseline is shown



(**b**) Difference between rscfl and baseline for 10K files



(**c**) Change in lighttpd latency histogram, 100K files. The shift in median expressed as a percentage of baseline is shown



(**d**) Difference between rscfl and baseline for 100K files

Figure 3.14: Resourceful overheads when running `lighttpd`

forming measurements. As in the case of the microbenchmarks in the previous section, we expect that the overheads will be percentually higher for serving small files. This is why I analyse two file sizes, 10K and 100K. Statistics about pages served on the internet today[23] show that html document transfer sizes between 6 and 10K are most common on both desktop and mobile. Average sizes for CSS files (77Kb) and Font files (123Kb) are equally spaced from 100K. Larger file sizes will be considered in later experiments, with 256K representing the size of a popular JavaScript library (jquery) and 512K being picked to represent the average total amount of scripts downloaded for web pages, as well as per-request transfer of video data (frames).

The results are presented in Figure 3.14. The graphics on the left (a and c) show a visual comparison between the two latency distributions; The median of the baseline is marked with a dashed line, while the median of the latency distribution with Resourceful measurements active is dotted. The difference between the medians is shown as a percentage of the baseline median. For small files, Resourceful increases the latency of very fast requests, but does not significantly slow down tail latency requests (the difference between the 99th percentiles is 3%). In general, the peaks of the baseline latency disappear: this is a direct effect of some requests slowing down and moving towards the right in the distribution, filling what were previously valleys. However, the general shape of the distribution is maintained.

The presence of the various peaks in the baseline `lighttpd` distribution is caused by the number of other requests a given request had to wait for: the first peak represents requests that were

[23]According to `http://httparchive.org`, statistics from Mar 2016, measuring data from the top 1 million websites according to Alexa rankings, with raw data publicly available.

served immediately, the second one requests that have waited for the completion of processing stages from one other request, the next one waited for two other requests, etc. This pattern will only be seen on small requests, as for larger requests and longer processing times everybody will wait on average on the same number of other requests, given by the concurrency level imposed by the arrival distribution.

Figure 3.14b shows the overhead distribution. From it, we're mainly interested in how far the peak is from 0. The fact that the distribution contains both positive and negative values is natural: it simply means that the two original distributions (baseline and rscfl) overlap. This just shows that the low-latency requests when running `lighttpd` under Resourceful have a lower latency than some high-latency requests measured while `lighttpd` is running normally.

For 100K files, the perceived overheads are even smaller: only a 5.13% shift in the median and an overhead distribution which is better centered around 0. Beyond the particular experiment that I have run here, I am interested in statistically determining the expected differences between the two samples, `baseline` and `rscfl`. For this, I run a Mann-Whitney U test on the two samples, under the null hypotheses that the distribution of `rscfl - baseline` is symmetric around 0 (i.e Resourceful introduces no overhead):

```
wilcox.test(lighty_rscfl$lat_cyc, lighty_baseline$lat_cyc,
            mu=0, conf.int=T, conf.level=0.99)

Wilcoxon rank sum test with continuity correction
  W = 5303600000, p-value < 2.2e-16
  99 percent confidence interval: [336208, 419816] (cyc)
  median difference in location :  377958 (cyc)
```

Listing 3.7: Results of Mann-Whitney U test for the 100K case at the 0.99 confidence level

The conclusion is that the null hypotheses can be rejected at significance level $p < 0.001$: The presence of Resourceful can be determined statistically. Furthermore, at confidence level 99%, the overhead of Resourceful is between 336208 and 419816 cycles (this represents between 5% and 6% of the baseline median).

This 5 - 6% latency overhead could be further reduced by targeting measurements (for example, only measuring cycles as a primary metric) or by limiting the number of kernel subsystems that are probed (for example, enabling measurements for the memory allocation subsystem will inevitably result in numerous cross-subsystem transitions, with the corresponding probes firing and adding overhead).

In terms of throughput, I have run repeated experiments (15 repetitions, 20000 requests each) and measured two throughput metrics: average requests per second (req/s) and average kilobytes per second (kbytes/s). This has been done for `lighttpd` without any changes (baseline) and for `lighttpd` with active Resourceful measurements, serving 256K files. Because of repeated sampling of averages, we can consider the distributions close to normal, and run a two-sample one-sided t test (under the assumption that the presence of Resourceful introduces overhead, and samples from that population should have larger values when compared to the baseline).

In both cases, the null hypothesis (that there is no difference in means between the baseline and resourceful measurements) can not be rejected (p=0.93). The 99% confidence upper bounds for the difference in means are:

```
99% confidence bound     baseline mean        rscfl mean
   [..   0.14 ]             447.8                447.7       req/s
   [..  36.51 ]           114741.2             114727.3      kbytes/s
```

The extremes of those confidence bounds show a 0.03% throughput decrease when compared to the baseline. Due to the small difference in means, more data would be needed to characterise the exact overhead in a statistically significant way. In the end, using Resourceful in production systems will depend on how much overhead is considered acceptable, and that level can be reached by introducing adaptive decisions of what to measure and when.

3.4 Understanding performance variability

So far, I have looked at the overheads imposed by either the proposed low-level probing mechanisms or by Resourceful as a whole. However, an important part in evaluating this framework lies in describing the provenance data it produces and in understanding how that can be used in diagnosing performance problems.

In showing how Resourceful can be used in diagnosing performance variability issues, I have picked `lighttpd` as the target example, because of the high importance of diagnosing latencies for networked services. `lighttpd` is simple and mainly targeted at serving static pages, but uses the same approaches as many high-performance client-server architectures, in that it is based on event loops, does request processing in multiple stages and multiplexes amongst them using epoll-like functions.

Therefore, `lighttpd` is representative of a larger class of services that are running both on premise and in the cloud today, which makes the following analysis a typical one. It also means that the methodology described here can be applied more widely.

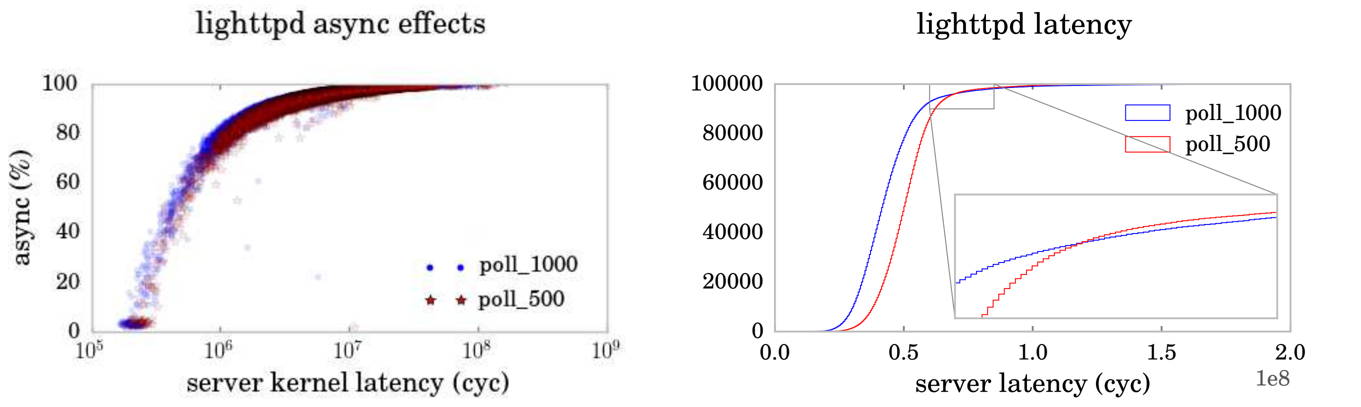
3.4.1 Asynchronous resource consumption

At the beginning of the chapter, I have addressed a number of issues that separate the way Resourceful does measurements from typical approaches used by the systems community today. Amongst them, the discussion about asynchronous effects. A reasonable question in this context is whether this asynchronous resource consumption really matters, or whether its magnitude is small enough to be safely ignored.

For `lighttpd`, I will study type 1 asynchronous effects (from the perspective of the measurement framework, the resource consumption happens before we know to whom we should attribute it to). This is mainly introduced in `lighttpd` by the use of `epoll_wait` calls. For the same web server, type 2 effects are much reduced, because the request processing, using `sendfile` on a non-blocking file descriptor is synchronous from an application's perspective (i.e if a call would block, the work is placed in an application queue and considered on the next iteration of the event loop). This introduces wait times, but the actual work is done synchronously, in a way which allows `lighttpd` to signal to the measurement framework what request is being processed at every instant. There is further asynchrony present in the TCP stack which is not tracked at this stage. However, the measurements are consistent with `lighttpd` considering a request complete as long as all buffers were handed for NIC transmission (data is queued for sending in the NIC driver).

Figure 3.15 looks at both the impact of changing `epoll_wait` configuration parameters on the final server-side latency and at the impact of the asynchronous effects as a percentage of the time spent inside the kernel. The choice of considering the amount of work done asynchronously in relation to the time spent in the kernel (and not to the whole server-side latency) is due to `epoll_wait` mainly doing work on the kernel side: in order to detect meaningful relationships we will need to ignore unrelated application side-effects on the latency, such as request queuing and multiplexing.

The `poll_1000` configuration uses the default `lighttpd` `epoll_wait` timeout of 1s, while `poll_-500` halves it to 500ms.



(a) The importance of type 1 asynchronous effects in relation to time spent by `lighttpd` in the kernel. The x axis is logarithmic.

(b) Server-side latency under two application poll configurations

Figure 3.15: The figures show how important are type 1 asynchronous effects to the latency of `lighttpd`. The difference between the two poll configurations is related to timeouts: poll_1000 uses an `epoll_wait` timeout of 1 second, while poll_500 reduces the timeout to half a second. In this experiment, `lighttpd` serves 256K files.

In the scatter plot shown in Figure 3.15a we observe that for low-latency requests, the impact of asynchronous wait times is insignificant: indeed, those requests were likely to have been processed immediately without waiting a lot of time on the `epoll_wait`. However, as the latency increases, the asynchronous effects clearly become dominant, with tail-latency requests doing almost nothing else on the kernel side *but* waiting on `epoll`.

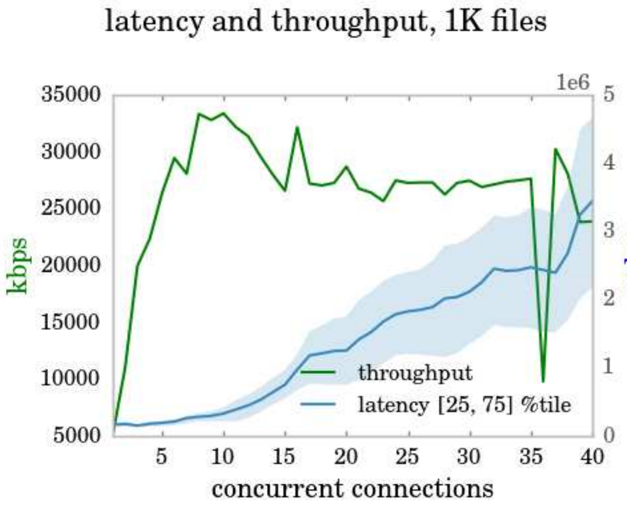
On the other hand, we see that tweaking parameters which control the asynchronous behaviour can have significant effects on the final distribution of latencies (Figure 3.15b): decreasing the `epoll_wait` timeout to half of its previous value slightly reduces the 95th percentile but at the cost of increased lower latency percentiles. The effect can be explained by considering that reducing the timeout will allow the progress of other request stages instead of blocking a relatively long time for straggler requests. However, the reduced timeout will also mean that other requests will have to pass through more loops in the event loop until their socket file descriptor receives new events.

Both observations lead to the conclusion that when doing per-activity measurements, it is necessary to consider asynchronous effects if targeting a detailed explanation of variations in latency. Ignoring or misattributing those effects to the wrong requests would introduce sufficient noise as to make attribution and root cause analysis significantly harder or even impossible.

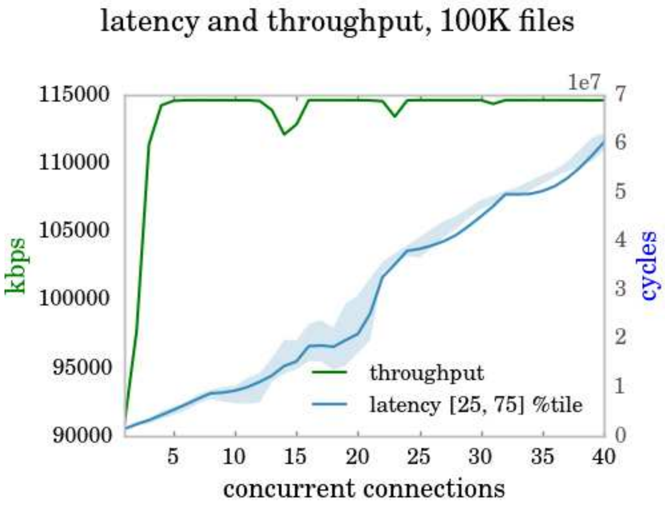
3.4.2 Latency breakdowns

When talking about diagnosing the causes of latency, it makes little sense to analyse an application well beyond the point where it reaches its maximum throughput. However, the region just after reaching maximum throughput, typically showing a "knee" in the latency curve, is interesting: Consider the case of `lighttpd` serving requests of a given size to an increasing number of concurrent clients. After the point where the web server reaches its maximum throughput, the latency will start increasing *because of queuing issues*. In other words, requests will start waiting more and more on the completion of other requests in the queue, although the location of such queuing is unclear (it could be on the application side, the kernel side – tcp stack, network card drivers – or in a busy router along the network path).

Resourceful can be trivially used to point out the place where such queuing happens (the bottleneck), if it is local to the operating system running the application (or, by exclusion, pointing



(a)



(b)

Figure 3.16: Single-worker `lighttpd` XR plots[25] defining the testing regime. For latency, the blue line represents the mean, while the shaded area covers values between the 25^{th} and 75^{th} percentiles. The throughput is the average over 10000 requests

to the network as the culprit[24]): one only needs to look at application-side request wait times, time spent in asynchronous wait (epoll) and time spent in the kernel's various subsystems. With this information, developers can decide whether the problem can be fixed by improving the way the server does multiplexing amongst concurrent requests, by optimising tcp stack parameters (or using a different tcp stack) or by modifying the kernel. Once such optimisations are performed and the performance remains worse than parameters in the SLA, the only remaining solution is spinning up new instances of the application and load balancing between them.

In this section, I will be focusing on a number of cases where the relationship between the latency distribution, application queuing and kernel-side operations is not immediately known, so further single-thread performance gains are possible. This implies I will not be interested in the multi-worker, load balanced case, but in diagnosing latency variability for a single instance of `lighttpd`, which uses a single-threaded event-loop for multiplexing amongst requests.

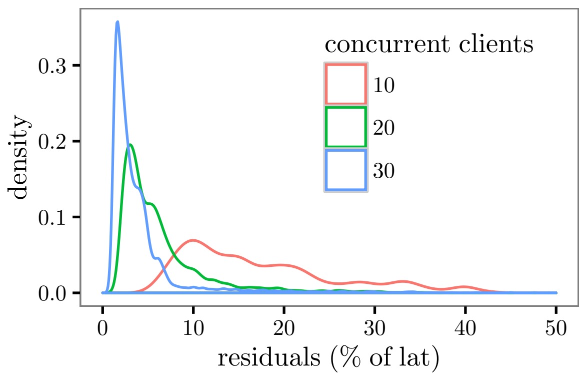
Furthermore, looking at the graphs in Figure 3.16, we will consider concurrency levels just after the maximum throughput is reached, while latency enters it's linear growth phase. For subfigure (a), that would mean between 10 and 20 concurrent connections, while for subfigure (b), it would be between 5 and 20. Generally, `lighttpd` reaches peak single-thread throughput from a low number of concurrent users. In both experiments, Resourceful is actively performing fine grained measurements; `weighttp` is used to generate the concurrent workload and runs on a separate physical server one hop away from the one running `lighttpd`. Below, I will show how Resourceful helps in answering detailed questions related to changes in latency.

Figure 3.16 also shows that `lighttpd` manages to sustain peak throughput even under load-test conditions and when the Resourceful measurement framework is active (while previously I have shown that the degradation compared to the baseline case is minimal).

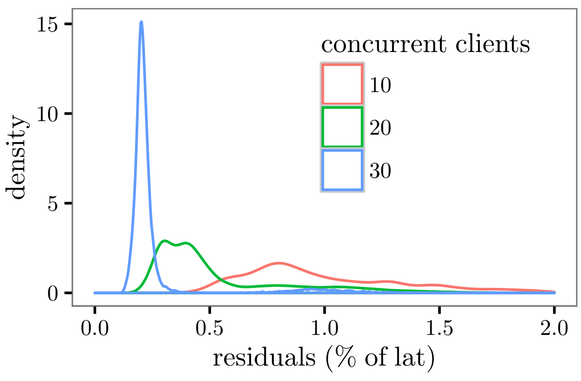
Are Resourceful measurements sufficient for providing latency breakdowns? Before being able to trust the analysis results extracted from Resourceful measurements, it is natural to ask whether

[24]Resourceful can be extended to run on network equipment with sufficient computing power available, but no such experiments have been performed so far.

[24]An XR plot characterises the latency (R) and throughput (X) as functions of the concurrency level



(a) `lighttpd` serving 1K files



(b) `lighttpd` serving 100K files

Figure 3.17: Residual server-side latency not explained by Resourceful measurements (typically representing work fully done in user-space by the web server) corresponding to the experiments in Figure 3.16

the collected data explains enough of the measured server-side latency to support the claim that it can be used in guiding system and application optimisations.

For this, we consider, for the same experiments giving the data in Figure 3.16, how much of the latency is *not* explained by Resourceful (in other words, the residuals), expressed as a percentage of the server-side latency. Figure 3.17 presents the distribution of results.

When serving 1K files, Resourceful data explains more than 80% (20% residual) most of the time. For the larger 100K requests the residual drops to under 2%. However, perhaps more importantly, as latency increases because of contention and queuing, the measurements capture an increasing percentage of the final latency (above 20 concurrent connections, Resourceful data explains over 90% of the latency even in the 1K file case). In other words, the data not captured by Resourceful remains mostly constant and does not significantly contribute to changes of the latency distribution under contention. This means that in the important cases we want to cover we will have all the data needed for determining the sources of variation in latency.

What-if scenarios I have made the claim that the design of Resourceful allows obtaining estimates in "what-if" scenarios. The example picked to prove that point concerns the experiments conducted in Figure 3.16:

What would the distribution of latency for 20 concurrent connections be *if* the time each request waits for others (the wait time) is distributed in the same way as it was when `lighttpd` served requests with 10 concurrent connections, while keeping all other things fixed?

This allows us to explore what would happen to the final server-side latency distribution under slightly different conditions than the ones experienced during measurement. If by changing the distribution of wait times the latency distribution for 20 concurrent connections becomes more like the one for 10 concurrent connections, we can conclude that wait times are an important factor that has triggered the observed latency changes.

For now, I will make a number of simplifying assumptions regarding what measurement variables we can target for such "what-if" investigations:

- the variable represents an additive component to the final latency distribution (this is the case of wait times, time scheduled out, as well as cycles spent in each kernel subsystem). A counter-example would be a variable representing the number of L2 cache misses experienced during each request.

- the variable is independent of others and affects server latency directly: changing it will not trigger changes in other variables that also affect the final latency distribution. A counter-example would be trying to ask what-if questions on variables like schedule-out time: changes there would also modify the distributions of per-subsystem measurements, which in turn change the server latency distribution.

Making inferences when the conditions above are not necessarily respected is discussed in more detail in Chapter 4. The gist is that non-independence needs to be considered such that changes in the variable targeted by the "what-if" scenario are propagated towards the related variables, and subsequently to the final distribution. Regression techniques are therefore required for determining inter-variable relationships.

In our case, the conditions are fulfilled by both the wait time and the per-subsystem kernel cycle measurements. The data provided by Resourceful in fact defines an empirical joint distribution between all measured variables. Considering the data stratified per number of concurrent connections and limiting the analysis to 10 and 20 such connections, we know the following distributions:

$$\begin{gathered} P_{concurrent\_20}(req\_size, lat, wait, async, k\_cyc...) \\ P_{concurrent\_10}(req\_size, lat, wait, async, k\_cyc...) \end{gathered} \tag{3.1}$$

We also assume that

$$\begin{gathered} lat_{20} = wait_{20} + others_{20} \text{ (additive condition) and} \\ P(wait_{20}, others_{20}) = P(wait_{20}) \cdot P(others_{20}) \text{ (independence condition)} \end{gathered}$$

Here, the subscripts of individual variables represent the number of concurrent connections under which the distribution was measured. From those, in the "what if" scenario described above we want to estimate

$$P_{est\_20}(req\_size_{20}, lat^*_{20}, \boldsymbol{wait_{10}}, async_{20}, k\_cyc_{20}...)$$

and in particular the $P_{est\_20}(lat^*_{20})$ marginal, where as a random variable,

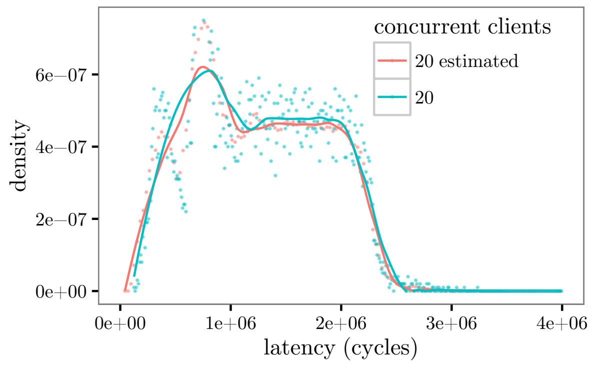
$$lat^*_{20} = lat_{20} - wait_{20} + wait_{10} \tag{3.2}$$

In terms of arithmetic operations between random variables, the first difference, $lat_{nw} = lat_{20} - wait_{20}$ can be exactly computed knowing the joint distributions in eqn 3.1 (by pointwise differences between the latency and wait times measured for each request).

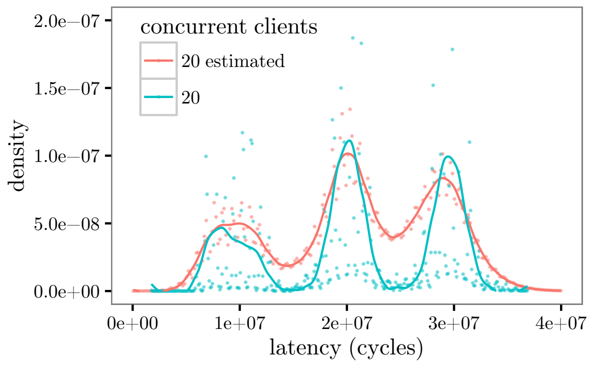
Adding $wait_{10}$ to lat_{nw} is the actual estimation step, which can be performed under the independence assumption using a convolution similar to the one discussed in the Methodology section when talking about determining overhead distributions:

$$\begin{aligned} P(lat^*_{20} = k) &= \sum_{i=-\infty}^{\infty} P(lat_{nw} = i) \cdot P(wait_{10} = k - i) \\ lat^*_{20} &= lat_{nw} * wait_{10} \end{aligned} \tag{3.3}$$

For testing the precision of the estimation in our case, I first run the what-if experiments by trying to add the $wait_{20}$ back to the latency distribution in the estimation step: $lat^{test}_{20} = lat_{20} - wait_{20} + wait_{20}$. The results are shown in Figure 3.18. For `lighttpd` serving 1K files, the recovered distribution is very close to the original, while for 100K files some "smoothing" takes place, but the overall shape of the distribution is maintained very well, showing the feasibility of the proposed method.

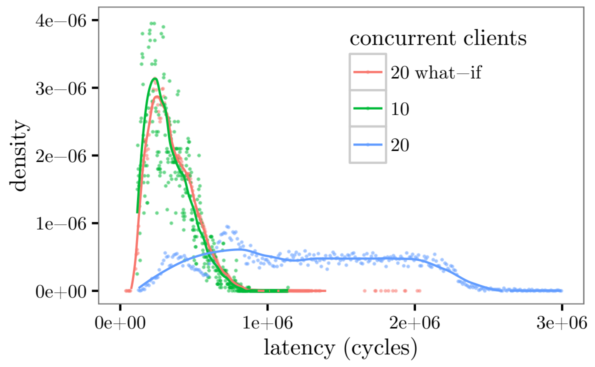


(a) `lighttpd` serving 1K files

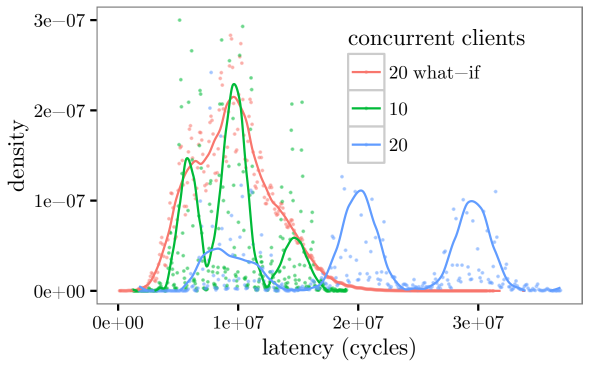


(b) `lighttpd` serving 100K files

Figure 3.18: Estimating the accuracy of the proposed "what-if" method, for the case of a known baseline. In this graph, we subtract the wait times from the server-side latencies $P(res) = P(lat - wait)$ knowing the exact joint distribution P(lat, wait). We then try to add the same wait times back without using the joint distribution.



(a) `lighttpd` serving 1K files



(b) `lighttpd` serving 100K files

Figure 3.19: What-if results: when considering a wait time distribution identical to the one measured for 10 concurrent requests, the 20 concurrent requests latency (blue) distribution approaches the one measured for concurrency 10 (green) – the red curve presents the estimation results

Applying it in the actual what-if scenario described in this section yields the results in Figure 3.19. They show that most of the difference in latency between the two cases can be attributed to application-side wait times: when considering requests running with 20 concurrent connections but waiting the same times as 10, the resulting range of server-side latencies are very close to the 10 concurrent connections case. This suggests the request multiplexing mechanism as a primary target for optimisation. In particular, we haven't reached the point of saturating the TCP stack or stressing kernel-side buffers.

We can continue identically by picking other variables for explaining the remaining differences. Furthermore, we can run similar what-if scenarios even with hypothetical wait distributions that were never previously measured.

The main factor that has allowed this type of analysis to be performed is the existence of fine-grained, precise, per-request measurements about resource consumption.

What causes the variability of latency to increase in Figure 3.16a? Let's consider the partial plot of Resourceful measurements in Figure 3.20 (containing all cycle measurements). On the x axis, the concurrency level increases as in Figure 3.16. We can notice the following things:

1. variations in wait time closely track the ones in latency, confirming what we have deter-

mined in the previous section.

2. `lighttpd` caches the small 1K files so no time is spent in the kernel block layer (`block_l`)

3. after 16 concurrent clients, more time is spent in the network, security and vfs subsystems; variability also increases after this point. The number of concurrent clients coincides with the beginning of a zone of constant throughput (so queuing starts happening on the kernel side even if the wait times are the ones that are dominating)

4. for 1K files, the resources spent asynchronously (waiting on `epoll`) is insignificant. This is caused by the requests being very small (the socket file descriptor will mostly be available for reading/writing). Most outliers appear in the region where throughput has not yet stabilised after reaching the maximum (up to 16 concurrent clients). This is yet another proof of queuing on the kernel side after that point: most of the time `epoll_wait` is called, it will return file descriptors which have pending events.

5. ignoring the tail portions of the distributions, the time spent kernel-side stays mostly constant for a wide set of concurrent connections

3.4.3 Conclusions

I have proposed a new approach in dealing with kernel-side measurements, starting from efficient low-level probing (kamprobes) and building a per-activity, real-time resource consumption measurement framework. The overheads imposed by this system have been evaluated using both microbenchmarks and a real-world application (`lighttpd`). In realistic scenarios, the expected increases in latency caused by Resourceful are between 5 and 12%, depending on the number of system calls executed per second by the targeted application. Further optimisations are available. The observed degradation in throughput is insignificant.

In terms of utility, I have shown that the data allows both looking at the elements that introduce variability in the system (enabling a targeting of optimisation efforts). Furthermore, developers are able to assess the behaviour of the system under new conditions by exploring "what-if" scenarios. Strategies for enabling such analysis for a larger class of systems and situations is discussed next.

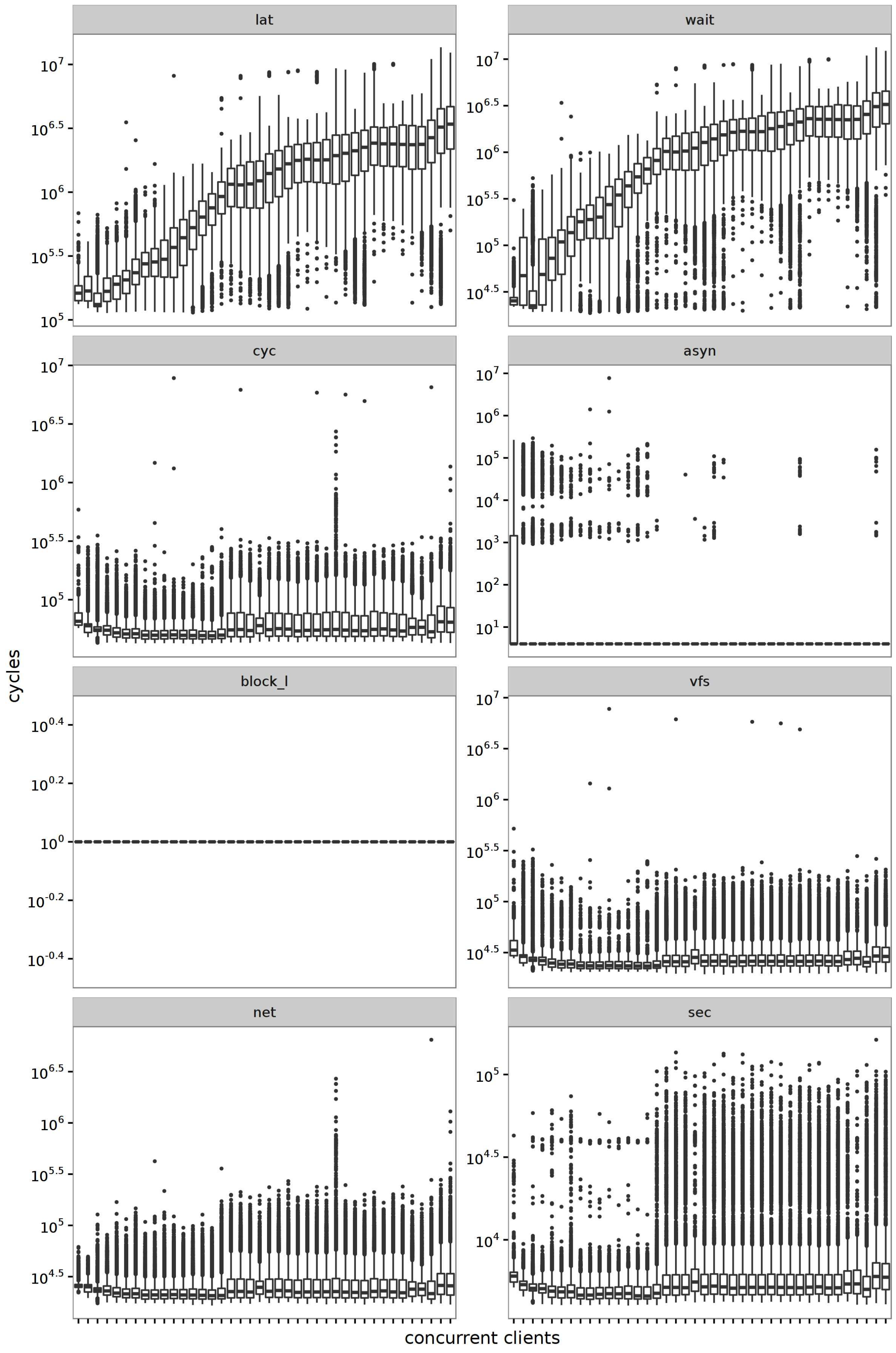


Figure 3.20: Resourceful measurement details, for `lighttpd` serving 1K files. The following distributions are displayed as boxplots for each concurrency level from 1 to 40: lat - server side latency, wait – wait time, cyc – cycles spent inside the kernel, asyn – cycles consumed during epoll waits, block_l – cycles spent in the block layer subsystem, vfs – cycles spent in the virtual filesystem subsystem, net – cycles spent in the network subsystem, sec – cycles spent in the security subsystem.

"The first principle is that you must not fool yourself and you are the easiest person to fool."

— Richard Feynman

4 Soroban: A provenance-based attribution framework[1]

IN THE PREVIOUS CHAPTER I have presented a number of low-level mechanisms that enable the collection of fine-grained provenance for different system and application properties (latency, resource consumption, errors). The aim was to explain unexpected variations of such properties taking into account OS-level resource usage and multiplexing. However, there is a need for defining clear methods for analysing and using this data, especially if we're targeting semi-automated diagnosis scenarios or systems that might change their behaviour based on such measurements. How would one design and implement those systems?

This chapter brings together the Resourceful work for obtaining data in the context of application activities with machine learning techniques in order to show how provenance-based computing could look like, discussing both system modelling and issue attribution based on fine-grained measurements. The main question remains one of understanding how provenance data can be combined with high-level, *local* user knowledge in order to make progress towards solving the use-cases described in Chapter 3.

The practical goal being considered is finding meaningful ways of comparing the behaviour of an application in two different settings: it could be a comparison between a program running on bare metal machines and it running inside a virtualized environment, alongside competing workloads; it could be the same software running on two different Linux kernel versions, or it running under different configuration settings. Similarly, comparisons could be done between normal states and situations like limited bandwidth, unresponsive communication partners, etc. Beyond measuring changes in behaviour triggered by conditions like the ones above, it is important *to be able to attribute their impact* on particular sub-components in our system. For instance, we want to be able to determine that "for this request with a high latency, the virtualization layer has imposed a 10ms overhead". Having similar cost breakdowns across multiple components should allow us to make informed operational decisions, as well as allowing for typical optimisation of bottlenecks. It is a case of knowing who to blame when the application does not behave according to expectations.

In the case of virtualization, the issue goes even further: customers should be able measure whether the characteristics of the virtualized infrastructure running their applications have met service level agreements (SLAs). Currently, most providers define such SLAs in a very coarse-

[1]An early outline of the work presented here has been published in HotCloud 2015 [Lucian-3], focusing on latency attribution in virtualized environments. The paper has been joint work with James Snee and Oliver R.A. Chick .

grained manner, typically referring to uptime, bandwidth or IOPS. However, opportunities exist for both sides if finer-grained prioritisation and resource guarantees would be available: customers that want to build complex applications on top of existing cloud infrastructure might be willing to pay extra for guarantees at the level of individual requests or classes of network traffic. Perhaps an application with a low average number of IO operations per second (IOPS) but with bursty characteristics will want guarantees about the service levels during such bursts, while it might accept slight degradation at other times. Others will be interested in memory bandwidth or NUMA placement guarantees.

If an application has components which are latency sensitive (i.e. state replication), or would like to offer more guarantees for business-critical clients, developers will want the ability to negotiate different SLAs for particular connections, cpu computations or disk transactions. Furthermore, they might want to set dynamic policies while also being able to clearly estimate predicted expenses. This type of fine-grained SLAs will require more transparency from the service provider, as well as good measurement tools on the client side.

As an example, the tools presented in this chapter will allow clients to determine how much of the latency of a connection can be attributed to slowdowns in the virtualization layer. Similarly, cloud providers have the incentive of running the same measurement tools to prove that eventual slowdowns were caused by application logic or failure modes. Some of those might have in turn been triggered by infrastructure issues, etc. Here, the provenance nature of data captured by systems such as Resourceful becomes clear: drilling down for causes and being able to perform attribution of consumed resources are just two types of analysis enabled by the same underlying data.

Although most of the work here is targeted at supporting those scenarios in practice, it is also important to develop the theoretical frameworks that can provide a rigorous setting for discussing about measurements, causality and provenance (as well as diagnosis in general). For example, such a theory would allow an understanding of:

- **efficiency: what should be measured and when** in order to incur minimum overhead but to maximise the information obtained from the system
- **inferencing: what kind of predictions or what-if scenarios are possible** given the current knowledge about the behaviour of the system
- **control:** determining what parts of the system need to change and how in order to achieve certain performance or cost targets
- **limitations: what questions can't be answered** with the current amount of recorded data

In this area, the use and extension[2] of Structural Causal Models (SCM), as described by Judea Pearl [112] provides the required tools for dealing with issues like the ones described above. While the development of statistical theory goes beyond the scope of this thesis, we will focus on a methodology for measurement that can be transposed in causal language.

4.1 Goals, approach and alternatives

The move towards cloud computing implies a transition to running most services on shared infrastructures, with less control over unwanted interferences and the resulting performance characteristics. The use of virtualization and hypervisors in particular introduces additional, uncoordinated levels of indirection with respect to process and VM scheduling. While the current mindset in the industry is that virtualization can provide strong isolation guarantees in terms of

[2]This is joint work with Philipp Geiger, from the Max Plank Institute for Intelligent Systems in Tübingen. Gathering experimental data and providing domain-specific knowledge in this space have been only my responsibility.

CPU and memory, one must understand this in the sense that each process in a set of collocated applications becomes slower by some degree when compared to its bare-metal performance[3]. In environments where the number of tenants changes dynamically, there are no guarantees that this slowdown is uniform. We must thus consider how it might affect the tail-end of service times or the end-to-end throughput: the performance is clearly influenced by the overall system state over which applications know nothing about (number of other services running on the same physical host, scheduler time slices or generally the contention on shared resources).

Despite previous work on improving scheduling in order to provide more isolation guarantees [36, 130], the argument above points towards the need of tools that can pinpoint the reasons of performance deviations at the level of individual application activities (one database query, responding to one http request etc.). In particular, such tools should help in ascertaining whether the observed anomalies are caused by the running service (e.g. garbage collection at application level) or if they are due to the external environment (e.g. CPU or IO starvation). This maps directly to the goal stated at the beginning of the chapter, of comparing different situations in which an application executes.

Current methods of characterising the behaviour of applications in a shared environment are focused on OS-wide measurements (`perf`, `iotop`), and rely either on benchmarking the VM and the application itself, or on continuous monitoring of OS and process metrics.

Benchmarks are not typically representative of production workloads [50, 126] and extracting useful information from them is challenging: no benchmark will simulate real world events such as VM creation, boot storms or major changes in load for particular tenants. Furthermore, benchmarks themselves do not provide optimisation routes or root cause analysis without significant expert knowledge and incremental trial and error.

On the other hand, root cause diagnosis systems based on monitoring data [107, 106, 88] have so far ignored the difficulties of trying to dissociate the resources consumed for performing an action from other activities happening concurrently. The need to do so, especially in regards to event-loop based programs, has been known since the idea of resource containers [8] was proposed. Without taking this into account, as evidenced in Chapter 3, one ends up measuring variables with possible hidden confounders. Thus, errors like not distinguishing correlation from causation, *attributing anomalies to the wrong cause* and ultimately making suboptimal resource allocation decisions become more likely. Because of this, using queuing models on top of data at the incorrect granularity, like proposed in CloudScope [34] can be unreliable in practice, even if there are instances where such an analysis gives the correct results.

With Soroban I propose a structured approach, in four stages (Figure 4.1):

1. Collecting fine-grained kernel level measurements that describe the resources consumed by application-level activities, using Resourceful.

2. Outlining a causal graph capturing the beliefs of developers/architects in regards to the existence of relationships between measurements or (more importantly) lack thereof. This guides the subsequent analysis, but it is accepted that the resulting graph could be incomplete or partially incorrect.

3. A training phase where we establish the ground truth in regards to the performance impact of making changes to the application or to the environment in which it runs. For example, we will run a web server under multiple client concurrencies, request sizes, number of concurrent VMs, competing applications, and under multiple configuration options. The training phase needs to exercise the system sufficiently in terms of independent variables in the graph (nodes without parents) for being able to learn typical behaviour. In situations

[3]in other words, isolation here means protection against unfair distribution of resources rather than against performance degradation

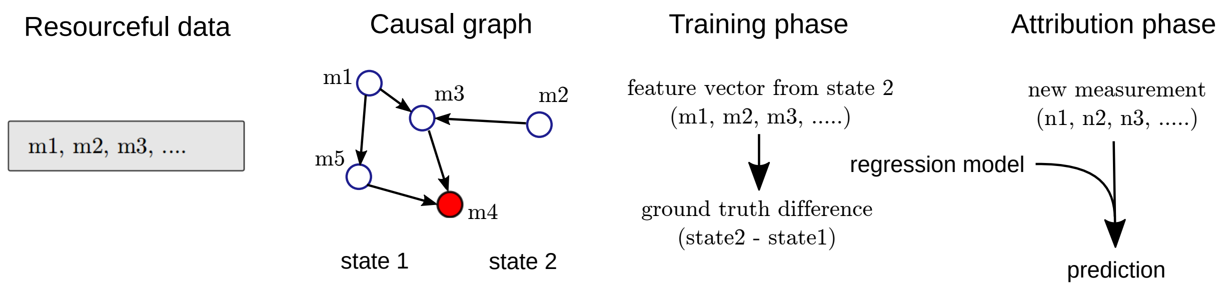


Figure 4.1: A general structure for Soroban inferences. In this case, we want to understand and attribute variations in one metric ($m4$) between the application in two states of the runtime environment (state 1 vs state 2). As an example, state 1 could be the application running on bare metal while state 2 could be the application running inside a virtualized environment. For each state, we gather the same type of measurements, as described by the causal graph. In the training phase, we build a regression model given a measurement feature vector and the ground truth difference of $m4$ between the two states. The data is stratified on random variables that have no parents in the graph (activity properties). Later on, given the regression model and new measurements in the system, we can predict how much of $m4$ is caused by factors in state 2 relative to state 1.

where the available search space of options is too large, sampling techniques will need to be employed. Alternatively, one might choose to learn from previous production workloads or operation conditions.

Assume that we want to understand the influence of running `lighttpd` under virtualization, when compared to the bare-metal case described in Chapter 3. We will run the web server in both cases (bare-metal and virtualization with increasing number of collocated VMs). For each of the per-request measurements realised under virtualization, we compute a ground truth metric representing the performance degradation[4] when compared to the bare metal case, for "equivalent" requests. A regression can now be done for learning the nature of the relationship between the set of measurements for a given request $(m1, m2, m3, ...)$ and the ground truth G_t. The resulting model can be then used in the next stage:

4. Attribution: the application can now be run in the target environment (in the cloud). For each application activity (responding to the http request), a prediction can be made with respect to how much of the observed performance variation is caused by the environment (virtualization) and how much is due to factors intrinsic to the application (high web server load).

The underlying assumption is that the fine-grained measurements will capture the aspects that vary differently between the considered states (i.e bare-metal/virtualization), making it possible to recover an estimate of the metric computed as "ground truth" difference. This metric could be as simple as a categorical classification (i.e "high cloud contention" vs "low contention due to the cloud"), it could represent parameters of the environment (number of concurrent vms) or could lead to exact estimates (i.e. how much of the latency of a given request is caused by the hypervisor when running under virtualization).

4.2 Design and implementation

4.2.1 Measurements and causal graphs

Building a causal graph for variables measured in a given system is very useful in encoding expert knowledge about the way the system operates, and an essential component of the analysis

[4]for example, latency or throughput

methodology I propose. The causal graph makes explicit the assumptions of an expert about the relationships that exist between measured variables.

graph consistency checking

However, due to system complexity, it might be challenging to obtain complete and accurate causal graphs. The existence of further causal relations, as well as possible spurious relations in the graph, needs to be checked against collected data. The graph can start out from the relationships determined based on provenance, and can be then augmented independently by people with in-depth knowledge about each layer (application, kernel, hypervisor, networking, etc). In this way, the graph allows the accumulation of local knowledge about how measured values are related to each other.

Of course, the description above does not cover the initial definition of such a casual graph. Essentially, this represents a modelling task as any other and should be done by experts in the various subsystems being measured. If an expert in virtualization (for example) has defined a subgraph modelling relationships between measurements taken inside a hypervisor, those can be linked as influencers in other subgraphs (for example, defining the behaviour of an application). The application level experts need to be aware of which of their measurements are affected by hypervisor-level variables and mark those influences explicitly. Variables that should be sums of their immediate ancestors in the graph but are also measured independently should be used as checkpoints to determine what percentage of the measured value can be explained using the causal graph.

As with other models, this suggest an iterative refinement model rather than being able to automatically determine (except for simple situations) the causality graph. In order to perform any attribution analysis, each set of application collected metrics will need to be included in a (possibly new) causality graph. However, parts of other causality graphs that define the relationships between external metrics (i.e metrics taken from a hypervisor, I/O subsystem or from the kernel scheduler) can be reused. When reusing part of a graph B in new one, A, experts should ask the following question for each of the nodes in B: “which of the metrics in A is directly changed if this node in B changes?”. Then, edges need to be added from that node in B to all of the influenced nodes in A.

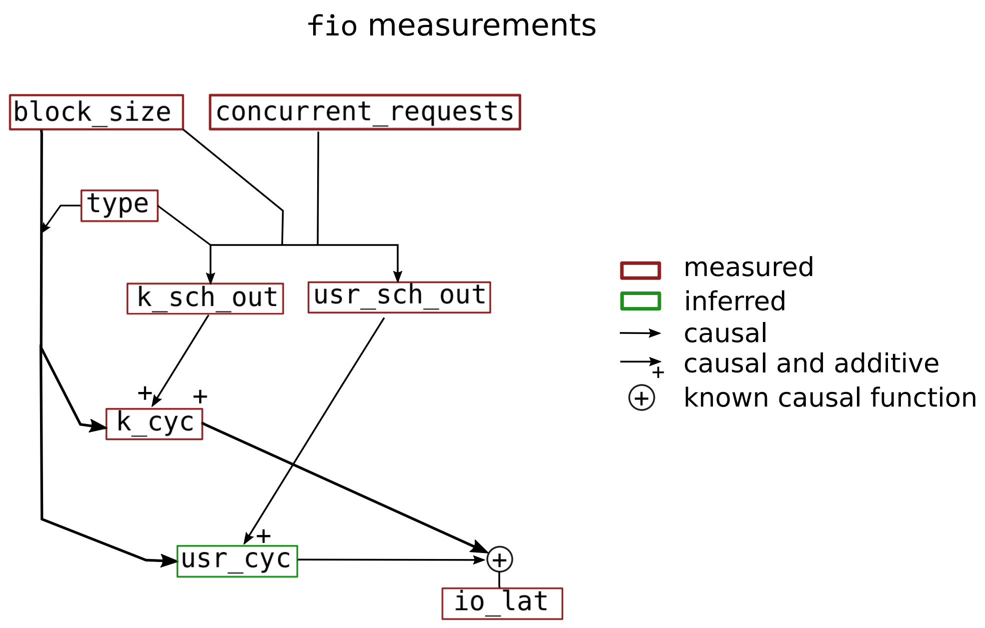
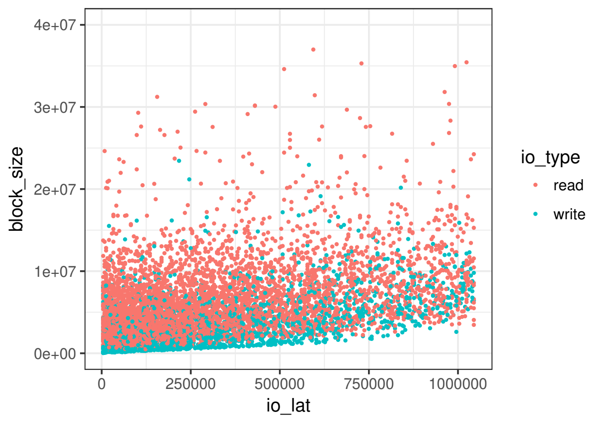
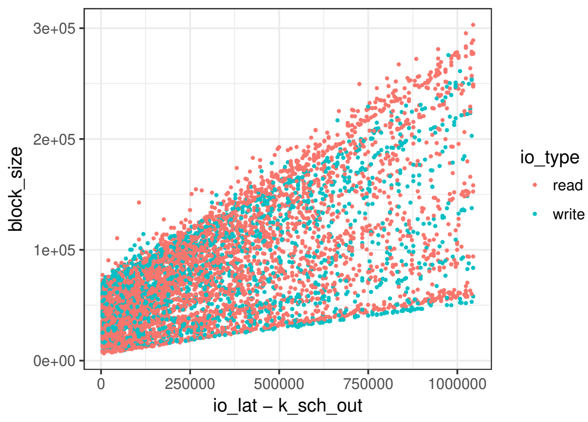


Figure 4.2: Causal graph for `fio` measurements Each node represents a random variable. The *lack* of an arrow between two variables means there is no direct causal relationship between them. Arrows that have a plus sign near them show that the parent is an additive component of the child (if the parent would be smaller by x, the child reduces its value by the same amount). Encircled nodes (i.e. $\oplus$) describe known causal relations: $\oplus$ means the child variable is the sum of all variables incident to the circle.



(**a**) fio normal correlation results for indirectly connected variables



(**b**) after considering relations in the causality graph

Figure 4.3: Causality graph sanity check: testing for the existence of stronger correlations after using information from the graph

To make the discussion concrete, I will start from a relatively simple example depicting the causal graphs for some of the Resourceful measurements done while running the `fio` I/O benchmarking tool. This graph is shown in Figure 4.2. For each value or application property that we measure, we need to decide what other values does it *directly* influence. In this particular case, our modelling considers that the `block_size` and the `type` (read/write) of the I/O request do not directly influence the latency of that request. Instead, they influence the time spent by the application in userspace (`usr_cyc`), the time spent in the kernel (`k_cyc`) and the corresponding preemption times: For example, a larger I/O request could lead to the the executing code path being preempted for a longer time while the disk is fetching the data, either in userspace (`usr_sch_out`) or inside the kernel (`k_sch_out`).

We have also made the decision that whatever latency is accumulated during preemption is automatically added to the time spent in kernel and user-space. Furthermore, we've *predict* that the final latency should be the sum of times spent, `k_cyc+usr_cyc`.

A direct link is added between two nodes only if the value of the parent has a *causal* influence on the value of the child. Causality in this context is understood to express the fact that intervening on the value of parent nodes affects the value of the child. The immediate parents of a node X are taken to be the only random variables (in the model) which directly influence the values of by X. Both direct and indirect (path) connections are important: for example, in the graph `k_sch_out` is connected to `k_cyc` directly: this states that `k_cyc` can vary because of `k_sch_out` even if all other variables stay fixed; on the other hand, the effect of indirect arrows can be eliminated through interventions: `concurrent_requests` has an indirect influence on `k_cyc`, but the graph tells us that this influence can be completely removed if we intervene in the system by fixing `k_sch_out` and `usr_sch_out` (for example, by running the application on an isolated CPU).

Of course, simply the fact that we've hypothesized such relationships doesn't make them true. However, based on them we can test a series of predictions: for example, the fact that by subtracting `k_sch_out` from `io_lat` we should see a stronger relation between the `block_size` and the final latency (`io_lat`), as it is just the block size that now keeps influencing the latency through `k_cyc`.

I test this by running a `fio` benchmark varying the block size from 4K to 1Mb, with a mix of 30% random writes and 70% random reads, using the psync backend configured to do `DIRECT_IO`. Four threads are doing I/O simultaneously and reading/writing 1Gb each. The results can be seen in Figure 4.3: it is very difficult to explain the relationship between block size and latency by looking at the data directly (a). However, we use our casual graph-based prediction and discover

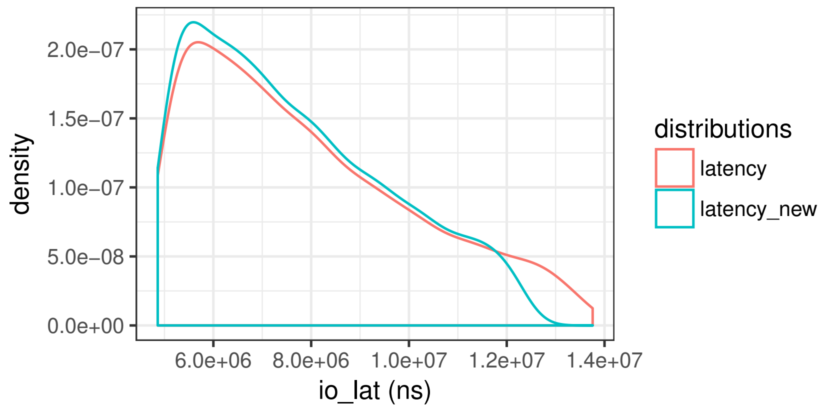


Figure 4.4: `fio`: original latency distribution (latency), and estimate after the schedule out time distribution is shifted by 5% (latency_new)

a stronger correlation pattern (b). We should interpret it in terms of explaining the remaining contributions to latency after eliminating a significant part of scheduler influence. For each block size, a "preferred" latency range exists, allowing us to make probabilistic predictions with respect to the final latency even if the scheduler behaviour changes.

For example, let us assume that because of an optimisation, we manage to shift the latency diribution of k_sch_out by 5% to the left, across all percentiles. That means that (for example) a value that was in the 0.25^{th} percentile is now in the 0.2^{th}. The following R script describes the exact data transformation, implementing a percentile - to percentile difference between the distribution and its shifted variant:

```
#find out the corresponding percentile for every data item
data$k_sch_out_percentiles<-ecdf(data$k_sch_out)(data$k_sch_out)

#shift percentiles by 5% to the left
data$k_sch_out_new_percentiles<-data$k_sch_out_percentiles - 0.05

# normalize data; more complicated estimates can be applied
data<-subset(data, data$k_sch_out_new_percentiles > 0)

data$k_sch_out_new<-quantile(data$k_sch_out,
                             data$k_sch_out_new_percentiles)
```

Listing 4.1: Quantile shift operations (R script)

After running those operations, we have obtained a transformed measurement variable, k_-sch_out_new, which we can add (according to the causal graph) back to the residual latency that was being explained in Figure 4.3(b).

For each data point, we are effectively computing (io_lat - k_sch_out) + k_sch_out_new and we thus obtain the overall I/O latency distribution under the new hypotetical scenario. The result is shown in Figure 4.4. Please note that this is not simply a shift of the original latency distribution.

If no clear relationship can be determined numerically between two elements that have been connected by an expert in the graph, a number of possibilities should be taken into account:

- The model is to coarse-grained and is missing one or more important variables; the connection is not direct but mediated by something else;

- The connection is indirect through some other variable already in the graph;
- The domain expert's assessment regarding the causal relationship was wrong, and the model should be refined by removing the link

Once some confidence is built in the correctness of a particular model, we can also understand the types of information that we would *not* be able to extract from it. For example, consider we run a second experiment with a fixed block size, performing just random writes instead of a mix of reads and writes. Based on the constructed causal graph, we know we will not be able to attribute differences in the latency distribution between this experiment and the original one to the block size or type of operation. This is because both those factors influence the exact same downstream nodes: we know there are no measurements allowing us to make the distinction between their contributions.

4.2.2 Transferring knowledge across causal graphs

Let's now consider a second example, which will be presented in detail for exemplifying a more complex attribution strategy. The corresponding causal graph is presented in Figure 4.5. It was built based on a detailed understanding of OS and Xen schedulers and their possible impact on a given workload.

Looking at the left side, titled "Local measurements", it's easy to see it is quite similar with the graph we had built for `fio`. In fact, that graph was describing a common pattern for the latency of an application activity that is being influenced by the OS scheduler. The `wait` and `conc` (concurrency) variables have been added because unlike `fio`, `lighttpd` multiplexes its application-level activities based on an event loop. A similar pattern should appear in any event-loop based application.

This means that it is possible, at least partially, to take and combine existing subgraphs when creating a causal graph for a given application and set of measurements.

4.2.3 The `lighttpd` causal graph

For allowing a better understanding of what is measured for experiments in this chapter, I'm going to describe the meaning of each variable in the graph; the nodes surrounded by red rectangles are determined from direct measurements. For requests sent by test clients to a `lighttpd` server, we will have access to the joint distribution of the random variables in the graph, representing the resources consumed for preparing responses. At times, it might be sufficient to measure a fraction of those variables in order to explain variations in application behaviour. However, we start with a maximal approach in order to understand what information is useful and needed (I also consider variable selection strategies based on mutual information).

`req_size` the size of the file requested by the client. In experiments, this is either kept fixed to a representative value (10K, 100K or 256K) or varied according to a given distribution. In production workloads, this would be decided by the distribution of client requests.

`local_load` within the VM running `lighttpd` (henceforth called "target VM"), this variable stands in for any resource-consumption activity that other applications besides lighttpd execute. This could be a periodic backup task doing IO, an application server running alongside the web server, or a database server. In experiments, `local_load` is a variable describing how many load-generating processes (`stress`) are run alongside `lighttpd`. This was picked (instead of a more accurate representation of load) in order to limit the amount

[5]The graph has passed through multiple iterations, and discussions with Philipp Geiger have contributed to its final form.

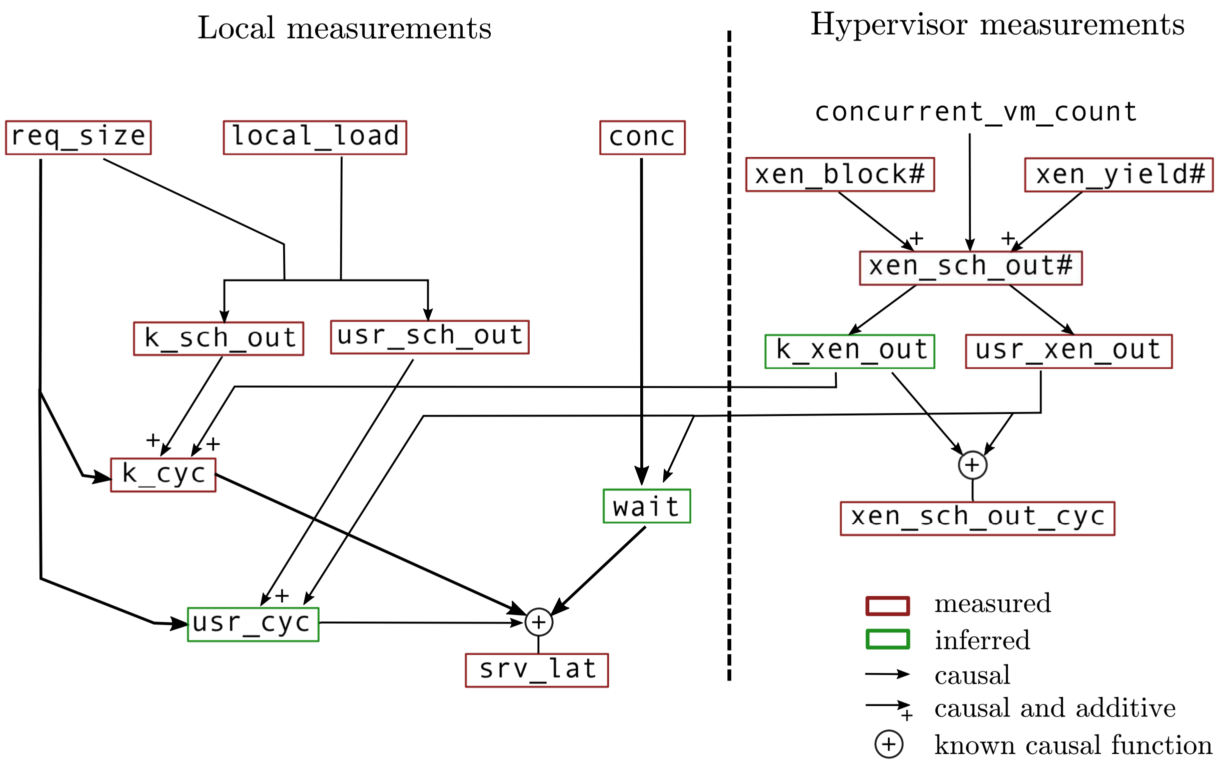


Figure 4.5: Causal graph for `lighttpd` measurements inside virtualized environment[5]. Each node represents a random variable. All the variables measured inside the VM using Resourceful are on the left side. On the right side, the graph contains variables representing hypervisor measurements. The legend is identical to the one in Figure 4.2

of information required when training a machine learning model, and help with its generalisation properties

`conc` concurrency level, counts how many requests were concurrent with the one for which the value is measured; This will either be constant when using `ab` as a test client, or vary according to the arrival rate distribution when using `weighttpd`

`k_sch_out` number of cycles accumulated while the kernel was preempted by another local task (those are switches that happen during system calls) for a given request

`usr_sch_out` number of cycles accumulated while `lighttpd` was scheduled out (while running in user-space) during the processing of a given request

`k_cyc` number of cycles spent in kernel mode for serving the request, as measured by Resourceful. k_cyc breakdowns for times spent in particular kernel subsystems are available but have not been represented to avoid cluttering the graph

`wait` cycles spent by a request while waiting for processing of other requests. This happens because the response is built in multiple stages, and those are multiplexed by the control loop. This is inferred by substracting the measured active time (time when we know processing has been done for a request) from the final latency (`srv_lat`)

`usr_cyc` cycles spent in user mode for serving the request

`srv_lat` the server-side latency of the request, measured from the time of the socket accept until `lighttpd` considers the request sent. Measured in cycles.

concurrent_vm_count number of other VMs running concurrently with the lighttpd VM on the same physical server. As a function of the running experiment, each such VM will run various load-generating tasks (using the same stress process as local_load) for generating realistic workload interactions between VMs. If different from stress, experiments will explicitly mention what workload is being executed on the other VMs.

xen_block# number of times the target VM has been scheduled out by XEN due to a block. For XEN, a block means the VM will not be re-scheduled again until at least one of the events for which the VM waits takes place.

xen_yield# number of times the target VM has been scheduled out by XEN due to a yield. For XEN, a yield means the kernel running inside the VM has decided giving up its time slice as it is waiting for a condition or event before being able to make progress.

xen_sch_out# number of times the target VM has been scheduled out by XEN, irrespective of cause (block, yield, expiration of time slice)

k_xen_out cycles the VM is scheduled out while its execution was in kernel mode, executing a system call from lighttpd.

usr_xen_out cycles the VM is scheduled out while execution was in user mode, running lighttpd. Here, lighttpd might have been processing either the target request or do work for one of the other concurrent requests on which the target request was waiting.

xen_sch_out_cyc total number of cycles the VM has been scheduled out during the srv_lat time.

The particular graph described by the variables above is geared towards understanding latency variations due to workload interference caused by CPU time spent executing competing tasks. However, the model can be easily augmented to cater for other types of interference (i.e. memory bandwidth or IO). It is simply a matter of adding such variables to the graph and configuring Resourceful to collect the corresponding data.

Also, as mentioned in the previous section, we shall stratify our experimental data on the following independent variables: req_size, local_load, conc and concurrent_vm_count[6]. Those define fundamental characteristics of each request and it wouldn't make sense to compare data which has different values for those variables in the training phase.

4.2.4 Xen changes[7]

Capturing the metrics present on the right side of the causal graph in Figure 4.5 (Hypervisor measurements) has required modifying the Xen hypervisor. In particular, the required changes were:

1. sharing an extra memory page between Xen and each VM (domain in Xen terminology). This page is filled by the hypervisor scheduler with details about when a domain is scheduled in/out and what has triggered that action (expiration of time slice, kernel request to block, kernel request to yield)

2. organising the scheduling events into a circular FIFO buffer, with the hypervisor maintaining the head (write end)[8]

[6]xen_block# and xen_yield# are also shown as independent in this graph. However, this is just because we have omitted some arrows in order to limit clutter. In practice, req_size, local_load, conc will influence them.

[7]Oliver R.A. Chick has implemented the first version of Xen data sharing, mapping extra memory pages from the hypervisor into each domain. I have done further optimisations, moving from a queue data structure to counter snapshots and have implemented the Resourceful-side token integration.

[8]this has been optimised to use simple counter snapshots instead of a circular buffer of individual events, but the underlying mechanism is simpler to explain when considering the buffer

3. allowing the domain(s) running Resourceful to maintain multiple tails (read ends) into the same shared buffer. In particular, as I explain below, each activity token maintains it's own tail pointer.

The main reason why coordination is required between the hypervisor and the VM-side Resourceful measurements is the fact that a given schedule-out event may affect multiple activities in the VM, even if those activities are not active at the time when the event happened. Taking the example of `lighttpd` multiplexing between various stages of two requests, with a Xen schedule-out event happening while processing request 1: even if the current active token is the one for request 1, the time the VM spends scheduled-out also contributes to increasing the latency of request 2. In other words, all concurrent requests need to account for this schedule-out event.

We implement this by allowing each token to maintain its own tail pointer into the Xen event buffer. Then, whenever the token is activated as well as on subsystem entry and exit we need to consume all the events between the token-maintained tail and the current Xen head pointer. The measurements are aggregated on the kernel side and added to the corresponding subsystem. A special subsystem, `XEN_Userspace` exists for holding the Xen events that have happened between application made system calls (in user space)[9].

Of course, these changes would not be available within existent cloud infrastructures. However, modifying the Xen hypervisor has two complementary justifications: firstly, we want to understand what types of information would be useful if exposed from the hypervisor side; if I can prove that even small pieces of information (such as how much time your own VMs have been scheduled out and why) allow application developers to make significant progress in understanding the behaviour of their application, then cloud providers might be convinced that being more transparent about such metrics is beneficial. Secondly, I also want to understand whether the information can be simply inferred from the client side. In that case, Soroban could work in unmodified environments and deduce the hypervisor metrics based on previous regression models.

An identical mechanism has been implemented by James Snee for measuring schedule in/out operations for Docker containers (this time, within a single Linux kernel). This proves that the strategy is general enough to be applied across virtualization solutions with very little changes in design.

4.2.5 Processing and plotting scripts

At the moment, Soroban predictions are not fully integrated into the Resourceful API. Therefore, applications can't directly make use of the diagnosis results in real time. However, for research purposes I have developed a set of python scripts that select the variables required for building a regression model, run the machine learning process (using `scikit-learn`) and allow for offline predictions on new datasets. Together with them, I have developed a set of plotting scripts (both in python and R) that have been used for creating all the figures in this chapter.

In principle, the prediction phase is fast enough to be run alongside slower requests or on a separate thread in the web server, informing the choice of multiplexing parameters (timeout time, priority) as well as scheduling decisions. However, experiments concerning those aspects are an area of future research.

[9]this is why the causal graph contains both the `k_xen_out` and `usr_xen_out` variables

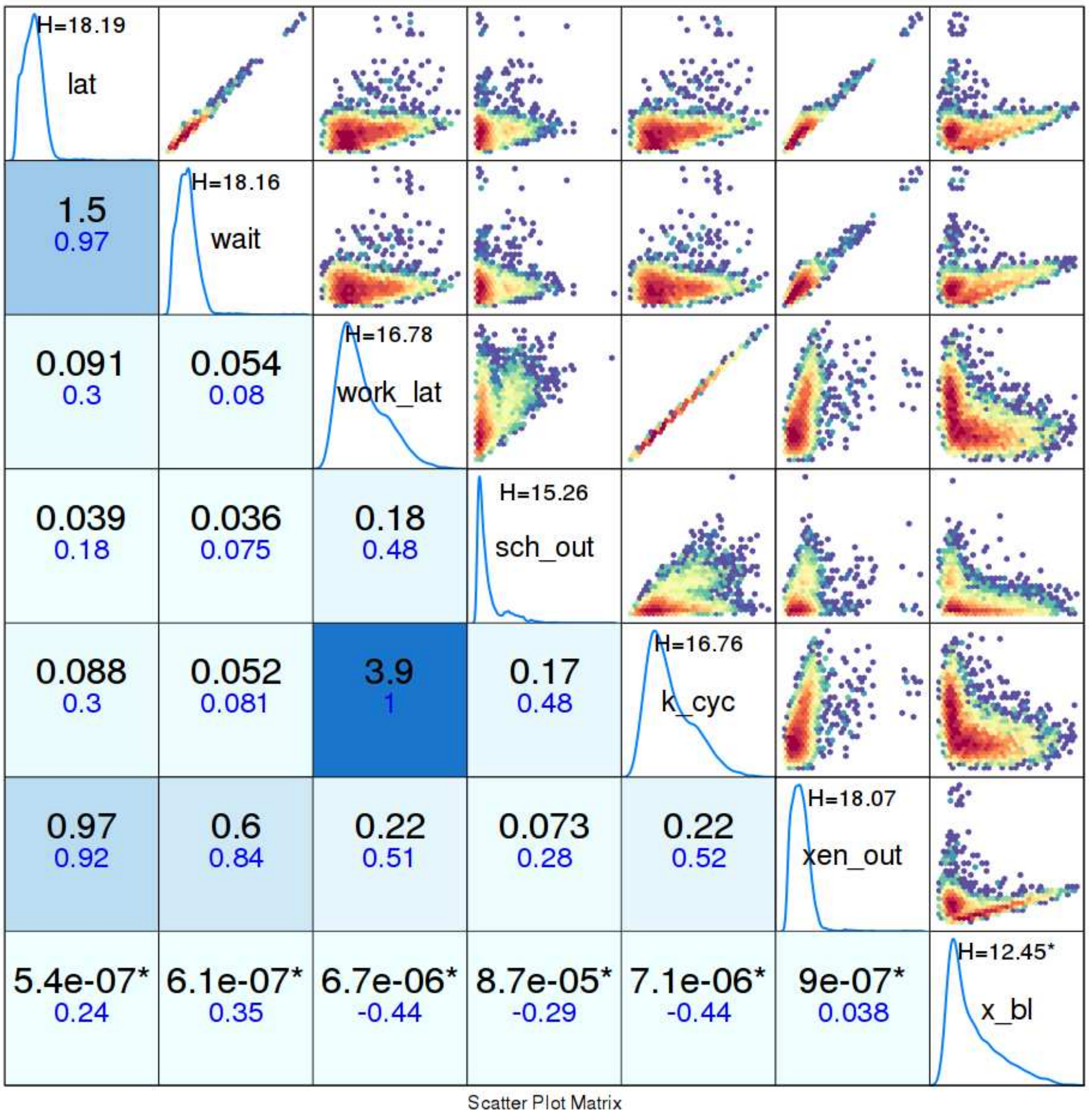


Figure 4.6: Matrix representing a `lighttpd` measurement "slice". The measurements were done while serving 256K files, with 18 other concurrent VMs running on the same server and running `stress` with 1 CPU worker and 1 IO worker. Above the diagonal, a hex-binned version of the scatter plot is shown, with color representing density from low (blue) to high (red) on a logarithmic scale. The diagonal shows the marginal density plots of each variable, together with the estimated entropy of that variable. Under the diagonal I plot both the mutual information between the variables (in bits, large font) and the correlation coefficient (small) for comparison.
The color of the cells under the diagonal is in accordance to the value of mutual information, with darker colors representing stronger relationships. Entropies and mutual information values followed by * have been computed using a plugin estimator because of the failure of the KNN method to converge, and might have larger bias.

4.3 Machine learning with provenance data

4.3.1 Variable selection for latency attribution

If one would simply plug all the measurements done by Resourceful into the regression model that is built in the training phase, unnecessary noise would also be added, reducing its predictive power. This is because certain variables might have nothing to do with the values we want to predict (i.e predicting the change in latency under virtualization). Including them when building the model is undesirable because the regression might pick up spurious correlations, so we first need to decide on a strategy for variable selection.

This is challenging because of two factors: (i) the nature of the distributions of the measured variables (non-normal, multi-modal) and their relationships (sometimes nonlinear), and (ii) the fact that such relationships might be hidden by other variables.

The first issue is problematic because it makes using correlation coefficients a bad measure of the relationship between two variables: we don't expect all relationships to be linear, and the inherent variability and measurement noise in certain variables would artificially reduce the value of the correlation coefficient even when strong underlying relationships exist. For an example of non-linear (and non-functional) relationships, it is sufficient to look at real data in Figure 4.6 (some variable names were abbreviated for space reasons[10]): consider the pairs (`k_cyc` ~ `sch_out`) and all relationships of `x_bl` with other variables. In the case of (`k_cyc` ~ `sch_out`), the scatter plot shows a composition of multiple functions (the effects of individual kernel subsystem measurements that have been aggregated into `k_cyc`).

To overcome the limitations of correlation coefficients, I propose the use of *mutual information*, a measure based on entropy.

When considering two variables X and Y, the mutual information, $I(X;Y)$ tells us how much knowing one of the variables is reducing the uncertainty about the other. If X and Y are independent, knowing Y does not give any additional information about X and thus their mutual information is zero. At the other extreme, if X is a deterministic function of Y, all information contained in X is shared with Y (the variable contains information that is in some sense redundant *if X is known*). In this case, the mutual information is the same as the uncertainty contained in X alone (the entropy of X).

The mutual information can be expressed mathematically as (we use $H(X)$ to denote the entropy of X):

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \end{aligned} \tag{4.1}$$

However, entropy and implicitly mutual information can be hard to estimate accurately for arbitrary datasets. This is why in terms of implementation I use a method of determining mutual information based on k-nearest-neighbours (KNN), which shows minimal bias [81] instead of applying equation 4.1 directly.

When making the selection of variables, we're not necessarily interested in the absolute value of mutual information; I propose using it as a ranking method for picking the 'top-k' interesting variables (with k set by a human user). For the example in Figure 4.6, when diagnosing latency we can look at the first column to see that `wait` and `xen_out` are primary candidates for inclusion into our model (the next one, at considerable distance, would be `sch_out`).

However, this first-pass in terms of selecting variables is not sufficient, because of the second issue (variables that hide the presence of relationships amongst other variables). Latency in particular is susceptible to this kind of problem because it is essentially generated by an additive process (each component in the system can *add* latency but not remove some). Going back to the causal graph, I have represented this type of relationships with a '+' annotation on various arrows and graph nodes.

To understand why an additive process poses problems in terms of statistical analysis, we can look at Figure 4.6 for the `lat`, `wait` and `k_cyc` variables. At first, the relationship between `lat` and `k_cyc` seems to be a very weak one ($I = 0.088$). The scatter plot also looks inconclusive.

[10]In the figure, some variable names have been abbreviated for space reasons: lat = `srv_lat`, work_lat = `lat - wait`, sch_out = `k_sch_out + usr_sched_out`, xen_out = `xen_sch_out_cyc`, x_bl = `xen_block#`

However, the causal graph defines wait as an additive component of lat. This means we can precisely remove its influence on lat by subtraction, using the fact that we know their joint distribution (this is the work_lat variable). After the subtraction, work_lat has a very strong relationship with k_cyc ($I = 3.9$), while others also increase their mutual information with the target variable (sch_out goes from $I = 0.03$ to 0.18).

Thus my proposal for variable selection is an iterative one:

1. select the top-k variables that have most *direct* influence on the target variable (latency), based on mutual information
2. if any of the selected variables have a purely additive contribution to the variable we're trying to explain, subtract them out, yielding a new variable R (this is work_lat = lat – wait in our example)
3. continue the selection process iteratively on R.
4. consider and select *categorical* or count variables that are parents of currently selected variables. Even though they might have a poor correlation with the target variable (taking x_bl as an example), they will likely help in distinguishing features between various regions within scatter plots (see Figure 4.11 for an example in the evaluation section)

A way of improving the variable selection process described above is to also consider removing non-additive contributions in step 2. Doing this is possible by computing conditional mutual information: In the causal graph, conditioning should be interpreted as a way of blocking the influence of certain arrows. For example, if we want to better understand the relationship between k_sch_out and k_cyc, we could condition on k_xen_out for a given req_size. In this way, we eliminate all the influence that parents of the k_cyc node exert, except for the one coming from k_sch_out. Computing the conditional mutual information follows the same line as equation 4.1:

$$I(X;Y|Z) = H(X|Z) - H(X|Y,Z) \tag{4.2}$$

The variable selection strategy can also be used to perform sanity checks on the proposed causal graph: for example, if a strong direct relationship can be seen in terms of mutual information but no corresponding arrow exists in the graph, an expert needs to decide whether it makes sense to add the causal relationship or not, updating the graph accordingly.

The following is another example of performing checks on the graph: in the figure, xen_out seems to influence lat and wait directly (the graph confirms this, as we have ignored the unmeasured variable usr_cyc). After subtracting wait, the information added by xen_out is reduced considerably (in the matrix plot, this sown at (work_lat ~ xen_out)). This is consistent with the graph, assuming the contribution of k_xen_out to k_cyc is small (this is true given the measurements). We conclude that the graph in Figure 4.5 passes a summary consistency check.

4.3.2 Attributing latency - computing ground truth metrics

Comparing the performance evolution of an application under different conditions requires picking one or more "target" metrics that are useful in attribution and diagnosis. In order to keep the discussion concrete, I will continue the running example of comparing lighttpd running on bare metal versus under virtualization and trying to blame variations in latency either on the application or on the hypervisor. However, the same discussion would apply for an application running under two configuration setups, on two different kernel versions (to determine performance regressions) etc.

Keeping in mind that the ground-truth metric we compute is the same as the one we'll be able to predict for each request, we have a number of reasonable choices:

- Picking an existing variable which we know we won't be able to measure in the target environment (the cloud), or that is expensive to measure there. An example of this would be the `concurrent_vm_count` variable: this would allow an application to predict, for each request it processes, how many VMs were running concurrently on the same physical server. The information could help, for example, in understanding whether the load-balancing groups of the cloud provider are placing client VMs on physical servers with widely different loads.
- Deriving a new per-request variable representing values of interest. When performing the derivation, we can use knowledge about *all the states* in which we've placed the application (i.e measurements for both bare-metal and virtualized environments) in a controlled experiment. For the latency case, I suggest that it is useful to compute the difference between the latency of a request under virtualization and a *similar* request with `lighttpd` running on bare metal.

If we've picked an existing variable as our metric, we can proceed directly to performing the regression step (training). This will turn into a classification problem if the chosen variable is discrete[11]. When computing a new metric (in our case, what we consider to be the "ground-truth" latency difference between virtualized and bare-metal scenarios), the process is less straightforward. This is because we typically have two experiments, E_{bare_metal} and E_{virt}, without knowing the joint distribution of variables between them. In a sense, for a given request served under virtualization, we don't know the "corresponding" request on the bare-metal side in order to substract their latencies for computing the ground-truth difference.

One may think of a number of reasonable approaches for estimating what the "corresponding" request is:

Average latency of same-type requests: Looking at the causal graph in Figure 4.5, in terms of local measurements (which exist for both E_{bare_metal} and E_{virt}), a request is characterised by 3 "root" variables: size, concurrency and local load. We can start by stratifying the dataset on those variables, and computing bare-metal latency averages for each group. For each request served with virtualization, we may then take the difference between its latency and the average latency of the corresponding bare-metal group.

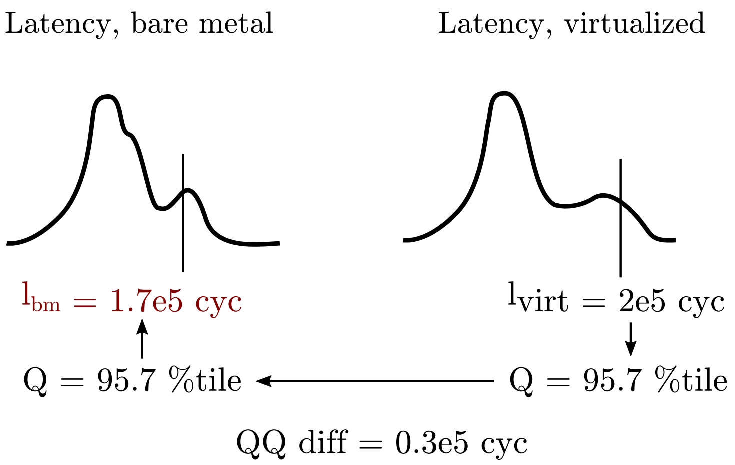
This poses a number of disadvantages: we have effectively summarised a whole distribution of latencies for each bare-metal group through a single number; furthermore we've seen previously that averages are not necessarily representative for the types of distributions that come up in practice. We could improve things by picking the median instead. However, quite a bit of information is lost: we will more than likely end up overestimating the true difference, as any request from the tail of the E_{virt} latency distribution will attribute its place in the tail to the fact that `lighttpd` is running under virtualization.

Naturally, this option is not useful in practice, but it highlights the elements that we must improve upon, namely the need to find "corresponding" requests which are in a similar region of the distribution as the request we're considering on the E_{virt} side.

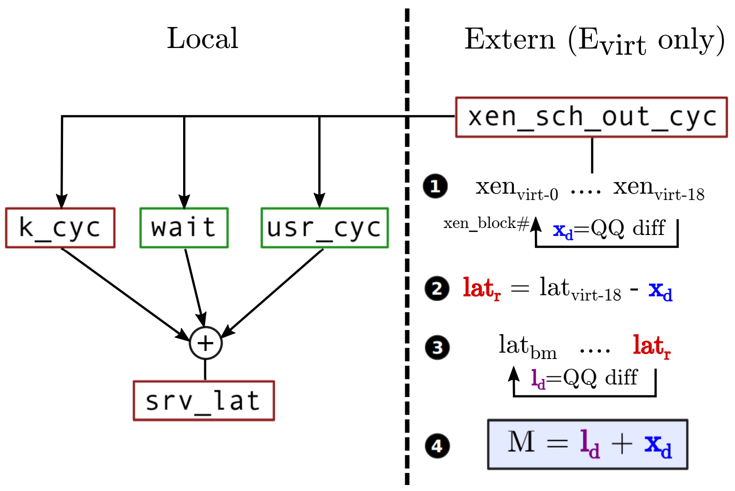
Quantile-to-quantile (Q-Q) differences: This strategy takes into account the relative rank of a given request within the latency distribution of each of the experiments. I start by observing that if E_{virt} would introduce no extra latency when compared to E_{bare_metal}, then all percentiles[12] of the two latency distributions would stay the same. Any shift in the percentile is obviously attributable to the changes we've made between experiments. Therefore, given the percentile p_{virt} of a given

[11] Gaussian processes can handle both cases

[12] Here, I will consider quantiles and percentiles as equivalent terms, without risk of confusion. In the typical definition, percentiles are 100-quantiles (the division of the probability distribution range into 100 intervals of equal probability)



(**a**) Quantile-to-quantile differences between virtualized and bare-metal latency distributions



(**b**) Iterated quantile-to-quantile, taking into account the causal graph. M is the resulting ground-truth metric

Figure 4.7: Computing ground truth differences

request latency l_{virt}, we can compute the latency at the same percentile in E_{bare_metal}, l^q_{bm}. Then, we can take $l_{virt} - l^q_{bm}$ to be the ground truth difference representing the latency that can be attributed to virtualization (Figure 4.7a). This is the approach that was initially proposed in the Soroban paper [Lucian-3].

The statistical justification for picking the corresponding quantile in a different distribution of the same metric is the comparison of items with equal probabilities: if the value I have measured in the virtualized environment is very high and highly unlikely (99.999 percentile), it makes sense to compare it with a similarly unlikely measurement in the bare metal experiment, to see the typical shift in the metric at that percentile.

Iterated quantile-to-quantile differences: Considering the previous Q-Q difference strategy together with the causal graph shows a potential problem. We essentially end up attributing the whole of $l_{virt} - l^q_{bm}$ as the contribution of the hypervisor to the latency. However, looking at the graph, l_{virt} contains two additive components: a local one (k_cyc, wait, usr_cyc) and one coming from the hypervisor (xen_sch_out_cyc) (Figure 4.7b). While the local component is influenced by the hypervisor, it can also vary while xen_sch_out_cyc remains fixed (because of the other variables in the graph). Now, let's imagine two scenarios leading to the same l_{virt}: one where the local component is large (for example, because of application-side wait times) and xen_sch_out_cyc is small, and the other where the local component is small (fast request from the perspective of the application) but xen_sch_out_cyc is large. According to the previous strategy, the same ground-truth difference will be computed. However, it is clear that we should assign a larger contribution of the hypervisor to the slowdown of the request in the second case.

What we actually want is separate predictions for the slowdown caused by the hypervisor under contention to the two components (local and external to the target VM). We can rephrase that as a what-if question: "*What* would the latency of this request be *if* the server would run on bare-metal (no external influencers)?". The solution to this problem shows how one can talk about what-if scenarios when relaxing some of the constraints we've defined in Chapter 3 (namely, the requirement of targeting only variables that are independent of all others and have a direct influence on the estimated variable). Furthermore, we are discussing predictions for particular measurements and not for entire distributions.

Consider the data slice presented in Figure 4.6, where lighttpd runs in a VM and there are 18 other VMs running on the same physical server. The VMs colocated with the lighttpd VM are running a synthetic stress workload using one CPU worker using all the CPU and one I/O worker continuously modifying a file and calling fsync, with a 66% duty cycle (2 seconds activity, 1 second wait). Because activity and wait times are not synchronised across VMs, this generates a noisy background workload. The process of determining the metric for a given request $req(lat_{virt\text{-}18}, xen_{virt\text{-}18}, ...)$ is shown in Figure 4.7b:

1. First we determine, for each external influencer, the QQ difference when compared to a baseline distribution, while stratifying the data in accordance to categorical or count variables which are direct parents in the causal graph. In our case, I pick xen_sch_out_cyc in the current distribution (18 concurrent VMs) versus the same variable in the non-contended virtualization scenario where the VM executing `lighttpd` is the only one running (virt-0). However, in virt-0 I will only consider requests with a similar number of xen_block#s and xen_yield#s. Any slowdown between bare-metal latency (l_{bm}) and $l_{virt\text{-}0}$ remains to be attributed to the local component of the metric. Result: x_d, the impact of contention on xen_sch_out_cyc.

2. Because the contribution of xen_sch_out_cyc to the final latency is additive (even through mediated by the local variables), we can subtract x_d from the latency of our request, $lat_{virt\text{-}18}$. Result: lat_r the latency of req if $xen_{virt\text{-}18}$ would have been $xen_{virt\text{-}0}$ instead (the latency without contention introduced by the virtualized environment).

3. Obtaining lat_r's quantile from the latency distribution in $virt$-0[13], we compute the QQ difference when compared to the bare-metal distribution. This gives us a measure of the slowdown of operations *inside* the `lighttpd` VM while under virtualization (the slowdown caused by the hypervisor to the local component of our ground-truth metric). Result: l_d, the impact of virtualization on `srv_lat`.

4. The final ground-truth slowdown caused by virtualization is obtained by summing up the local and external components: $M = l_d + x_d$

If needed, multiple ground-truth metrics can be computed, but in the following sections I will assume that a regression model will be built for each of them, separately.

We now have all the elements required for running a regression for modelling the relationship between the set of selected variables and the ground truth metric. I propose using a Gaussian process for this purpose, due to its good robustness in working with noisy data. The benefit

[13]The subtraction in point 2 has brought us from the $virt$-18 to the $virt$-0 latency distribution

of running this regression is that the resulting model will have some generalisation power over situations not encountered during the training phase, by identifying trends in M. Thus, it is better to build a regression model as a source of metric predictions (together with confidence intervals for their values) rather than storing all the raw data and applying the strategies above in determining an estimate of the chosen ground-truth metric on every request.

4.3.3 Gaussian processes training[14]

The problem I'm trying to solve is finding a function f that connects the set of selected variables $\mathbf{X}$ (section 4.3.1) representing measurements of a given application activity (in my example, `lighttpd` serving a web request), to the ground truth metric y (latency that can be blamed on the hypervisor, discussed in section 4.3.2). Unlike when computing ground truth metrics, $\mathbf{X}$ can *only* contain data which can be measured in the target environment (the cloud). Learning function f from the data is naturally formulated as a multivariable[15] regression problem, and Gaussian processes are a good fit for this task as they don't place inherent limitations on the distribution of the underlying data, and allow for measurement noise. As I have shown previously, most of the distributions encountered when doing `lighttpd` measurements are non-normal. Additionally, it is possible that the relationship between them and y is nonlinear.

For solving the problem, the Gaussian process considers a random variable F representing a general family of functions f_i with distribution $p(f)$, that are regression model candidates. The property imposed on F is that any linear combination of samples (functions) has a joint Gaussian distribution. In typical notation, we take $F \sim GP(m, K)$ to mean that F is distributed as a Gaussian process with *mean function* m and *covariance function* (kernel) K.

The kernel has significant control over the shape of the functions that will be sampled from F. In the experiments I conduct in the following section, K is chosen to be a squared exponential (Radial Basis Function). RBS has a number of hyperparameters such as noise variance, length-scale (determining how fast a function can vary) and roughness. Optimal values for those are obtained directly from the data, through an optimisation step.

The initial considered model, including n_i as the measurement noise term is:

$$\begin{aligned} y_i &= f(\mathbf{x}_i) + n_i \\ f &\sim GP(\cdot|0, RBS) \end{aligned} \tag{4.3}$$

This is updated by computing the posterior on f after seeing the training measurement data and corresponding ground truth values $M = \{(\mathbf{x}_i, y_i)_{i=1}^{n}\}$.

The resulting model is flexible enough for being able to weigh the importance that different dimensions in $\mathbf{X}$ have on the value of y locally. For example, it allows for the number of VM blocks to predict the hypervisor impact at low request latency, but it can place a higher weight on a different dimension (time the VM was scheduled out) at higher latencies. Once f is known, y can be predicted for measurements $\mathbf{x}_*$ that have not been seen before.

4.4 Evaluation

4.4.1 Setup

I run the experiments on the same physical server and under the same conditions as the ones in Chapter 3. The only difference is that instead of running `lighttpd` on bare-metal, I execute

[14]Dr Ramsey M. Faragher has helped in picking Gaussian processes as the right regression tool and tuning its parameters for Resourceful data, based on his previous experience in this area. I have developed the strategy for selecting variables and determining ground-truth influences taking into account the causal graph relationships, together with the scripts required for building the regression model and doing predictions on new measurements.

[15]some statisticians do not make the distinction between multivariable (multiple input variables, a single predicted response y) and multivariate (multiple input variables, multiple output variables))

it in a VM running on top of the Xen hypervisor. A modified version of Xen 4.6 is used, and experiments are run under two Xen scheduler configurations (`credit`, `credit2` schedulers). The only additions to Xen are for supporting the disclosure of VM scheduling data, as discussed in section 4.2.4. The `lighttpd` VM has one vCPU, and is allocated 7Gb of RAM. Resourceful measurements are only performed within this VM. Resourceful has special support for reading the shared pages exposed by the Xen hypervisor, but no further application-side changes are necessary (the data measured in Xen is added to Resourceful measurements as an extra kernel subsystem named "EXTERNAL_HYPERVISOR_XEN")

In the training phase, I run `lighttpd` under a fixed ab[16] workload (512K files, concurrency 20) but progressively increase the number of other VMs running on the same physical host. The workload is chosen as to keep a balance between work done in user-space and kernel-space while maintaining correspondence to real-world scenarios. Each of the other VMs have 1 vCPU and 512Mb RAM. I only consider the situations where at least 7 other concurrent VMs are started, as running a total number of vCPUs smaller than the available number of physical cores (8) introduces no contention (there are minimal observed differences between running 4 other VMs and running 7 of them). Each of the VMs runs a synthetic stress workload, using

```
stress -c 1 -i 1 --timeout 2
sleep 1
```

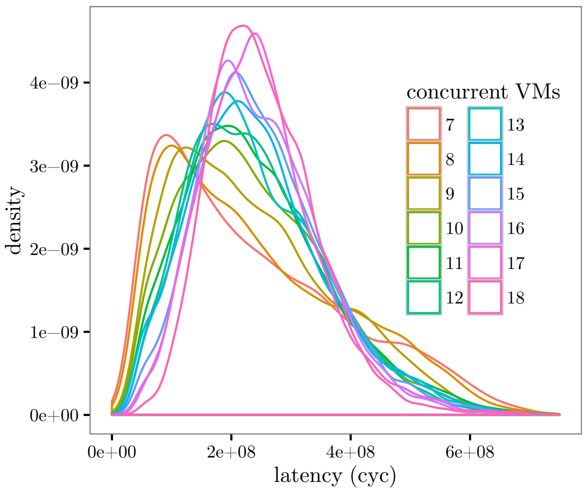
This means running 1 CPU intensive task and 1 IO intensive task (spinning on `fsync`) for 2 seconds, followed by a 1 second sleep (a 66% duty cycle). This setup is one that can be used in the training phase of real systems for which diagnosis is desired, if production workload characteristics are not fully known: by going from low contention to high, while exercising the main areas of resource consumption, Resourceful data will implicitly record the changes in application behaviour that are important for doing attribution.

Picking two sets of Xen scheduler configurations is done for providing some context of what can be expected in terms of contention introduced by Xen under different conditions, but also for highlighting an important aspect in the training process and a limitation of Soroban: if the underlying mechanisms that generate a given ground truth metric for the impact of the hypervisor change, previous models of attribution become obsolete. A comparison between the behaviour of the two schedulers which justifies the observation that the underlying latency-inducing mechanisms change is made in Figures 4.8, 4.9.

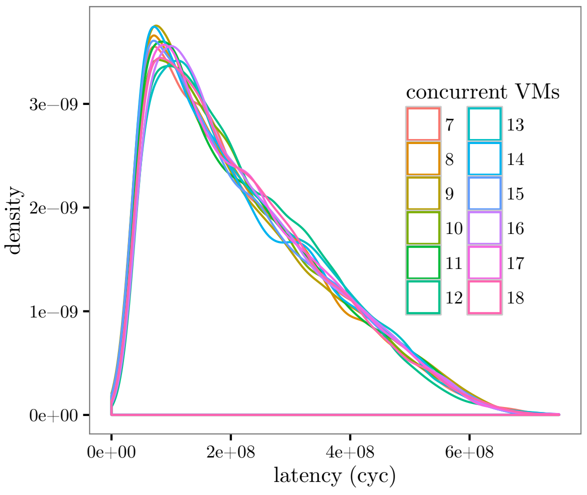
4.4.2 Training: the case of two hypervisor schedulers

Under the test workload, the `credit2` scheduler (which is optimised for low-latency applications and higher VM density) introduces much less variation in the presence of contention. In contrast, the `credit` scheduler introduces a clear shift towards the right of the distributions as contention increases (Figure 4.8a). This shift towards the right can be observed in most of the metrics, including the latency induced by kernel operations (Figure 4.9a) or by scheduling (Figure 4.9c). In the latter case, the distribution also changes its shape, with the appearance of a second peak towards the tail. No such changes can be observed in the individual components that create latency in the `credit2` scheduler case (b). Here, the final latency distribution becomes wider, with local measurements (k_cyc, k_sch_out, usr_sch_out) staying similar to the uncontended case of the `credit` scheduler, but overall variation as contention increases is minimal. This shows that the `credit2` scheduler provides much better isolation for the levels of contention tested here (2.4 VMs per CPU).

[16] ab is a HTTP benchmarking tool developed by Apache. I use ab as a load generator for the instrumented `lighttpd`. Single-threaded `ab` is not ideal for saturating server throughput, so any tests involving throughput testing use the `weighttpd` tool instead.



(a) Credit scheduler



(b) Credit2 scheduler

Figure 4.8: Evolution of request latency distribution as the number of concurrent VMs is increased. Here, `lighttpd` is serving 512K files on 20 concurrent connections. Under the load imposed, the credit2 scheduler provides a much more consistent median latency at the price of slightly wider distributions.

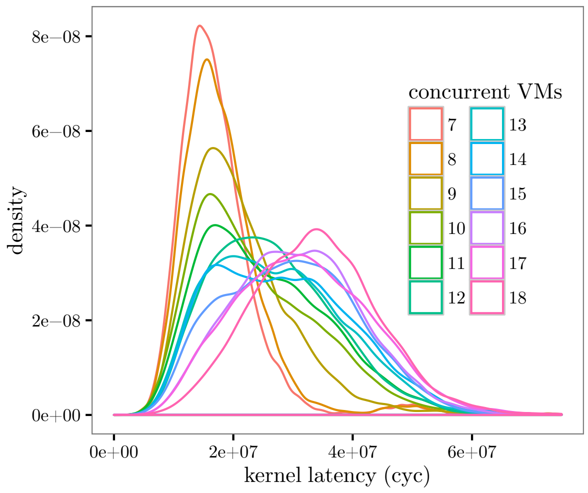
Furthermore, looking at Resourceful-measured data, we can describe the way in which the scheduling mechanism changes between the two schedulers (Figure 4.10). When using the `credit` scheduler, the hypervisor schedule-out time during each request is dominated by VM-exits (no blocks or yields but high `xen_sch_out_cyc`). This means that the VM running `lighttpd` is forcefully preempted in order to run other VMs, irrespective of whether it is running latency-sensitive tasks. In contrast, for the `credit2` scheduler, higher schedule-out times are correlated with an increase in the number of explicitly-requested blocks (the VM waiting for file descriptor events). This means that by better managing time slices and credits, the scheduler can schedule the VM out only when it can't make further progress, avoiding increases in latency. Because the mechanisms that determine the final latency are different between two system states (changing the scheduler), one will need to re-run the training phase the first time the scheduler is switched, or alternatively train under both configurations from the start. This does complicate the analysis and training phase, and shows that for obtaining accurate predictions developers must cover at least all the configuration/state changes which can not be directly inferred through regression.

If a developer trains on 3 or 4 request sizes, it is reasonable to assume that the behaviour for *similar* request sizes can be predicted. However, the same is not true for changes of schedulers, garbage collectors, hypervisors or categorical configuration parameters which pick different data structures or algorithms for the running application. Those need to be explicitly varied during the training.

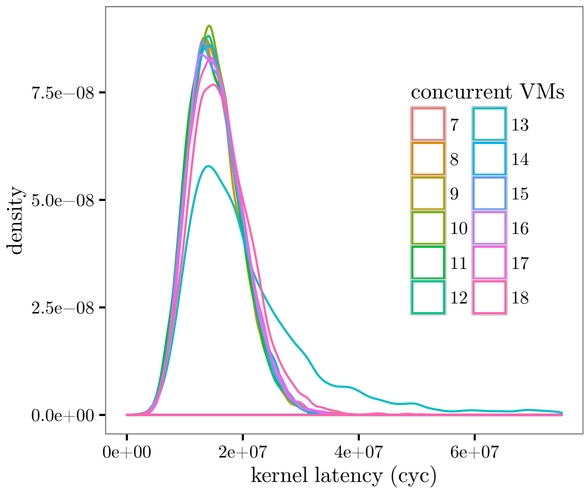
From the perspective of doing attribution (explicitly understanding the contributions of each subsystem or configuration change to the latency), the `credit` scheduler is more interesting, as there is more variation that can potentially be attributed to contention or poor hypervisor allocation of resources. This is why in the following analysis I prefer to discuss this scheduler instead of the more recent, optimised one (the `credit2` scheduler is in the beta testing phase). However, the exact same analysis can be applied to both configurations.

4.4.3 Ground truth blame: iterated quantile-to-quantile

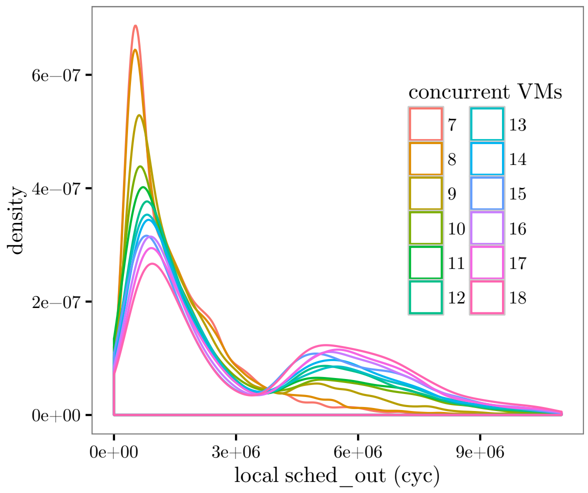
Before applying the Gaussian process regression, it is useful to understand how the proposed ground truth metric for determining hypervisor blame (i.e. how much latency is caused by contention in the hypervisor layer) looks on the training dataset. In order to better highlight how the



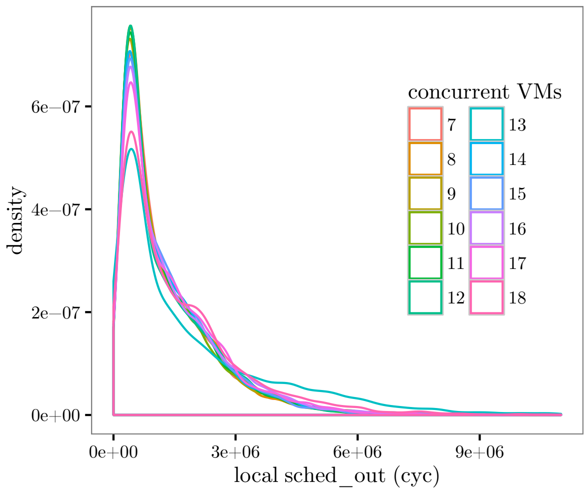
(**a**) Credit scheduler



(**b**) Credit2 scheduler



(**c**) Credit scheduler



(**d**) Credit2 scheduler

Figure 4.9: Evolution of kernel-side latency distribution (`k_cyc` in the causal graph) in (a),(b) and of time scheduled out `k_sch_out` + `usr_sch_out` (c),(d) as the number of concurrent VMs is increased.

proposed strategy works, Figure 4.11 shows a slice through the data, stratified by the number of concurrent VMs running on the same host as the `lighttpd` VM. The scatter plot matrix presents the evolution of the time that requests served spend being delayed by other VMs.

Under the proposed scheme, we need to compute 2 hypervisor blame metrics, x_d and l_d (as defined in section 4.3.2). The first one, (x_d) represents the influence of the xen scheduler in slowing down the request. For determining this, I compute the quantile-to-quantile difference between requests with a similar number of blocks[17] for a given number of concurrent VMs (from 8 to 18) and the base case (7 concurrent VMs). The difference expresses, for a given `xen_bl#`, the amount by which the distribution of cycles scheduled out shifts under contention. As an example, Figure 4.11 shows how the xen scheduled out distribution for a low number of blocks (purple) grows as contention is increased. Thus, the computed quantile-to-quantile difference will also grow accordingly. The results are presented in Figure 4.12: on low contention (9 VMs), the computed metric assigns a small part of the time scheduled out as xen scheduler blame. The proportion decreases with an increasing number of blocks. There are a small number of requests with a high number of blocks which are faster when compared to the base case: those

[17]this implies a clustering on `xen_bl#`, the number of VM blocks encountered during the processing of the request

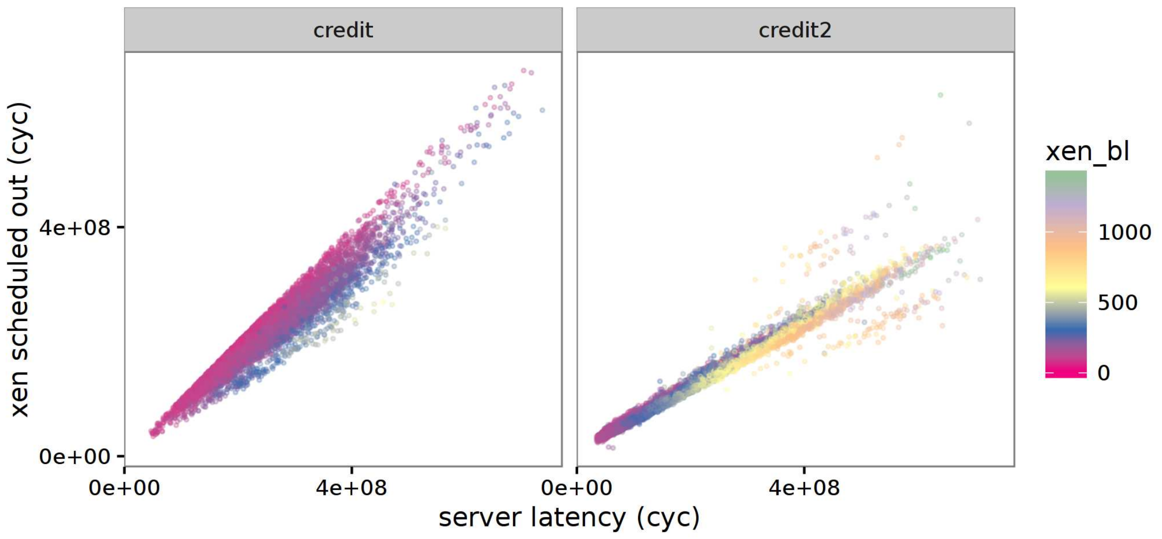


Figure 4.10: Example comparing the behaviour of different Xen schedulers under contention, for `xen_sch_out_cyc`. This considers 18 concurrent VMs running alongside the `lighttpd` VM.

are synergistic with the adaptive credit-based scheduling algorithm, and end up being marginally faster by blocking frequently and not consuming the whole time slice.

The case is similar for the high-contention case (18 concurrent VMs). However, here the requests which don't block attribute almost all of the time being scheduled out as slowdown caused by the xen scheduler.

The second blame metric, l_d, aims to attribute virtualization blame for the remaining latency, $lat_r = lat - x_d$. The second quantile-to-quantile difference is taken between lat_r and the latency in the dataset measured on bare metal (no virtualization). The difference (slowdown) can then be attributed to the virtualization itself (slower I/O due to passing through Dom0, interrupts, hypercalls).

As can be seen in Figure 4.13, the fastest requests get assigned a negative blame. This is an artifact of Resourceful only measuring kernel-side operations within the guest OS. Some of those are faster than on bare metal because part of the I/O stack is running in Dom0. In reality, the whole operation (guest OS + time spent in DOM0 doing I/O) is slower than the bare metal measurements. However, I accept the negative blame as a compromise of not having to apply further changes to Dom0 (which could prove impractical in real cloud provider scenarios). Furthermore, this predominantly affects low-latency requests, while a typical system analysis will first target the high-latency requests in the tail of the distribution.

In principle, a second specialised Resourceful kernel module could run in Dom0 and share data with the other Resourceful modules running in guests for solving this issue. A second observation is that at this level (in contrast to x_d), more blocks translate directly into a larger blame: this is because those blocks all imply hypercalls and processing that is done in Dom0 (I/O stack, network stack).

The whole ground-truth hypervisor blame is then taken to be $x_d + l_d$.

4.4.4 The regression model

I perform the Gaussian process training[18] phase using the data presented above, selecting the following input dimensions (variable names are the ones used in the causal graph from Figure 4.5): `req_size` (data stratified according to its value), `srv_lat`, `k_xen_out`, `usr_xen_out`, `xen_block#`

[18]python processing script using `scikit-learn` version 0.15.2

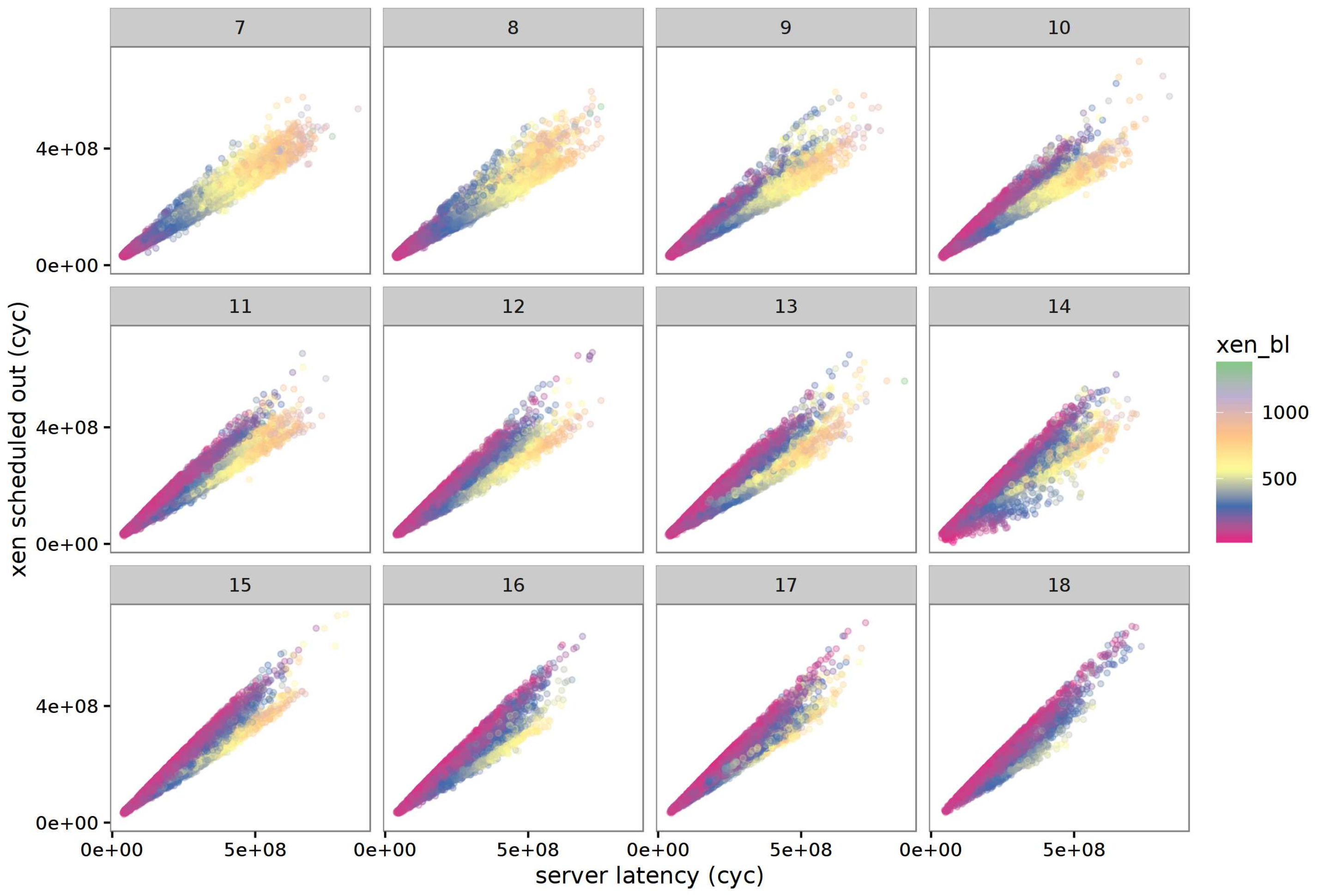


Figure 4.11: Evolution of the (`xen_sch_out_cyc` ~ `srv_lat`) scatter plot for the VM running `lighttpd` as the number of concurrent VMs increases from 7 to 18 (the title of each panel). Coloring represents the number of blocks (`xen_block#`) encountered during each request. In the uncontended case (7 VMs) an increased latency and time scheduled out can be explained by an increased number of blocks. However, as contention increases, most schedule-outs stop being voluntary: it is the hypervisor who needs to preempt the target VM for running the other VMs. Therefore, when `xen_sch_out_cyc` is high and `xen_bl` is low, the blame can be assigned to the hypervisor. The Gaussian process is used exactly for learning such associations automatically and for determining their importance in affecting metrics like latency.

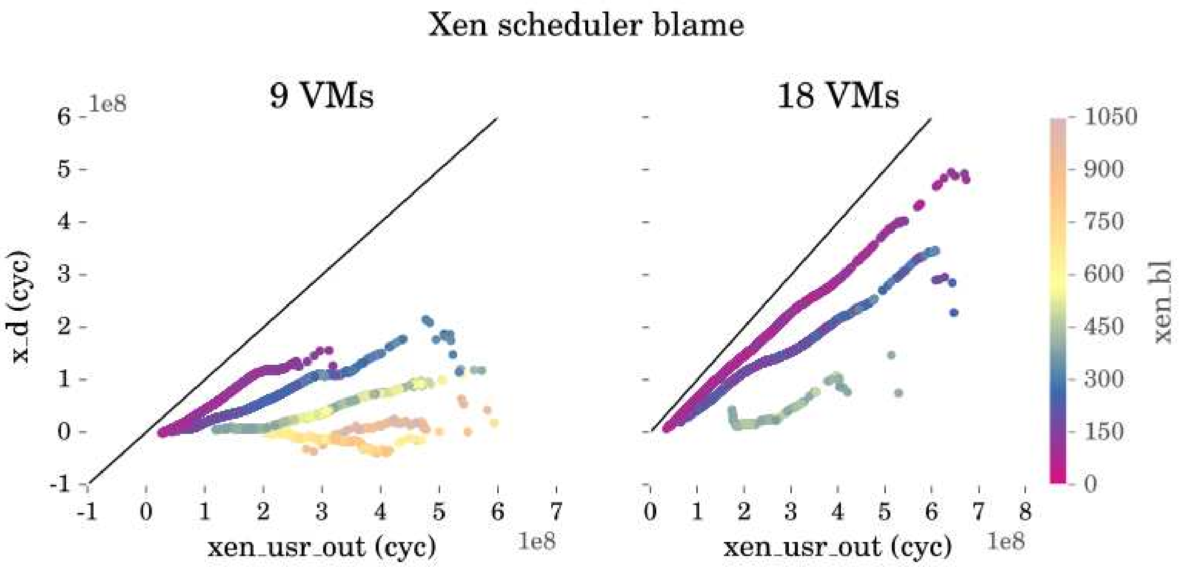


Figure 4.12: Xen scheduler blame component (x_d), quantile to quantile differences for each xen_bl group. Two contention states are presented, for 9 and 18 concurrent VMs. The solid line is $x = y$ (the extreme case where all cycles the VM is scheduled out can be attributed as a request slowdown caused by the xen scheduler)

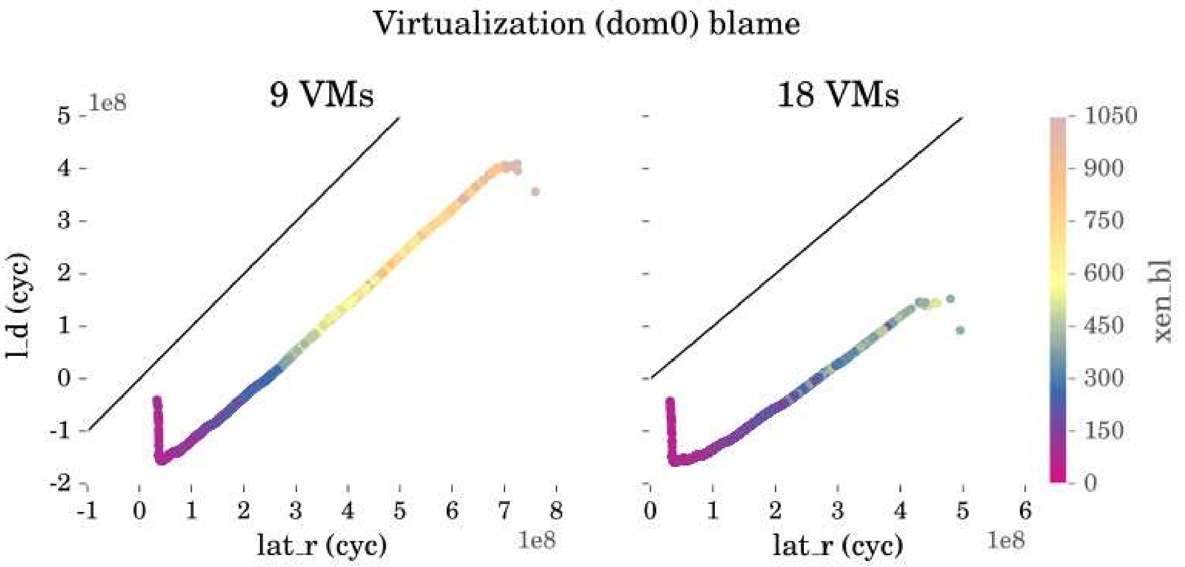


Figure 4.13: Virtualization blame component (l_d), quantile to quantile differences. Two contention states are presented, for 9 and 18 concurrent VMs. The solid line is $x = y$ (the extreme case when all the remaining latency can be blamed on the virtualization)

and xen_yield#. The learned variable y is taken to be the ground truth blame as computed above ($x_d + l_d$).

A regression model is built, relating the multi-dimensional vector of inputs (measurements) to the blame variable. Each prediction based on this model also returns the expected mean squared error (MSE), which is used for computing confidence intervals. The result showing slices through different dimensions of the model is presented in Figure 4.14. The overall trend is to associate higher blame values with requests that have higher latencies and more time spent scheduled-out (xen_sch_out). However, the contribution to *tail latencies* starts to be shared between the hypervisor and the application (seen as a lower xen blame in the graph). As expected, the shape of the model closely follows that of the ground truth metric (shown in section 4.4.3).

Based on the data so far, the model can be validated against ground truth blame. For the training process 1700 samples were taken across all numbers of concurrent VMs (from 7 to 18). 70% of this data was used to train the regression model, while 30% of it remains as a validation set. Data in the validation set contains both predictions from the regression model and the ground truth blame metrics. Figure 4.15 looks at the prediction error over this validation set. The residual blame axis of the figure shows how far away are the predictions from the ground truth in the validation set. Errors are an order of magnitude lower than the metric for which attribution is desired (latency), and the ratio stays similar in relation to the blame values themselves. Furthermore, there are no clear trends suggesting errors are affected by the region in the input space for which prediction is made (looking at the two projections in the figure suggests that the error doesn't increase with larger latencies or time the VM is scheduled out).

4.4.5 Attributing latency to Xen

Having validated the model, it is time to look at its utility: can the predicted data be used in performance diagnosis?

Figure 4.16 shows how Soroban perceives increased loads in Dom0. For this experiment, `lighttpd` runs in a VM while 15 other VMs (with the same load as I've used before) run concurrently on the same physical hardware. Additionally, a light load is applied in Dom0, executing `stress -c 1 -i 1 -d 1`. This means that there was 1 worker process stressing the CPU, 1 I/O worker spinning on fsync and 1 worker continuously writing to disk a 1Gb file. The Dom0 is pinned on 4 physical CPUs, so the load should by no means saturate its resources. The Soroban analysis is performed based on the regression model built in the previous sections. For each request served by `lighttpd`, I predict the contribution of Xen (together with the workload running on the concurrent VMs) to the server-side latency. I then compare those predictions to others from an identical experiment where no Dom0 load was present.

The scatter plot shows that in general the predicted blame is higher when the load runs in Dom0. However, it is important to notice that not all requests are affected equally: low latency requests behave pretty much the same as in the experiment with no Dom0 load. However, the load has a significant effect in inducing tail latency: the slow requests become significantly slower *because* of the load present in Dom0. In practice, this could be used to detect the presence of cloud administrative tasks (such as running anti-virus scans or performing periodic backups) that can slow down applications in ways that are hard to predict otherwise.

In the figure on the right of 4.16, we look at the quantitative analysis in this case: under the Dom0 load, the blame distribution has shifted to the right. More precisely, the 95th percentile blame has increased by 21 ms. This means that according to Soroban, the Dom0 load has contributed with an extra 21ms to the latency at the 95th percentile (156ms). Expressed as a percentage, this means a 13% increase.

4.5 Limitations of discussed methods

What-if questions on variables incommensurable with the target (explained) variable In the current thesis, I have only covered cases where the variables that enter what-if scenarios have the same units of measure (or are easily convertible to the same unit). For the previous considered example, this was the case as both the variable for which we were trying to do attribution (latency) and the variables we wanted to change under the hypothetical scenario (`xen_sch_out_cyc`, `wait`) were measured in cycles.

I haven't discussed categorical variables (for example, server configuration options) or other counts (i.e number of cache misses). i.e "how would latency change if I run a java application with -Xmx[19]=10M instead of -Xmx=1G?" or "what would the latency distribution look like if the cache misses incurred per request would be half of the ones observed?". In order to allow a detailed analysis in those cases, well-defined relationships between the categorical variable c (i.e. configuration option) and the target variable (i.e. latency) or another variable which is a parent of the target in the causal graph must be established. If all possible values of c are known, then a separate regression (`c ~ target`) could be performed. In the case of count variables, one would sample possible values of the count and attempt a similar regression. However, if the scatter plots between (`c ~ target`) do not show the possibility of predicting the value of the target as c changes (lines parallel to the target axis intersect the points of the plot in multiple areas) then the what-if analysis on c will remain impossible until the causal graph is refined to contain sufficient information to allow such predictions.

If the number of categories is small (i.e we're only interested in a particular what-if scenario and not in the ability to change the distribution or values of c arbitrarily), one might think to run the full procedure suggested by Soroban, comparing the system in the two situations "normal" and "hypothetical". However, this defeats the purpose of an what-if scenario, as the hypothetical situation must be actually measured and trained on.

Nonetheless, those observations still leave us in a better position than before, as they give criteria for determining that more information about the system is needed in order to answer the questions that we've posed.

Requirement for training stage The need for performing a training step before being able to use Soroban predictions of the metric chosen as ground truth for the difference between two states of the application is a limiting factor of the current solution. It means the application needs to first run in a controlled environment, with an experiment that changes system load parameters and workload distributions in a well-defined way. I envision two possible solutions to this problem: first, incorporating the controlled experiment in the continuous integration strategy used when developing the application: for example, policies might be added to perform the training step for each release-candidate branch with a particular tag.

The second possibility is using data captured by Resourceful while running the system in production or testing environments as training data - this is closer to a randomised experiment, but gives less flexibility in choosing the baseline state against which we compute the ground truth metrics. For example, instead of getting a prediction about the influence that a hypervisor had in slowing down a particular request when compared to the bare metal case, we would get the prediction compared to the state of the cluster when the dataset was captured. DevOps teams could choose data from when the application performed as expected as baseline, and select data for the other state (i.e virtualized with increasing stress) at moments when the level of contention is known or can be estimated. From this, the prediction would be able to later provide information on whether contention in the cloud can be blamed for any unexpected degradation in performance.

There is also the option of continuously computing the "ground truth" metrics as described in Section 4.3.2 instead of using a trained model. However, because we base our computations on quantile-to-quantile differences for a limited number of measured variables, the predictions will be affected by latency distributions that change dynamically over time (due to periodic workload changes, boot storms, anti-virus checks performed by the cloud provider etc.) The regression model essentially had the role of constraining the predicted values to the other observed variables, and this is lost when computing the ground-truth directly.

Requirement for a causal graph The task of building a causal graph for all measured variables in a complex system should not be underestimated. It will typically require a fair amount of

[19]-Xmx sets the maximum Java heap size

effort and insights into the details of each application and runtime environment. Naturally, such graphs will also evolve as applications suffer changes. Ideally, processes for invalidating nodes or subgraphs when changes in the underlying software are performed should be developed – but this is an area of future research. The presence of a rigorous testing harness when developing the application can help in automating part of the process of adapting the causal graph and re-running Soroban training phases when changes are made to the underlying software.

On the other hand, Soroban requires at least an approximate graph that can be validated against measured variables. It is hard to evaluate how missing links or variables will affect the accuracy and usefulness of provided predictions. Despite this, as long as the boundary between the two states (in our case between local and hypervisor measurements) is modelled correctly, the results should remain useful.

Subgraphs can be built and refined independently if they can be abstracted through a single node in a higher-level graph. In other words, if one knows what edges are incident to a high-level node, then the node can be replaced by a subgraph which models the behaviour of the system it represents more accurately. That being said, inferences like the ones described in this chapter are not feasible on top of higher-level graphs, because representing multiple processes through a single measurement will result in inter-variable relationships which are hard to predict using regressions. Applying clustering in the scatter plots and then doing regressions for each pair of clusters might work, but the procedure reverts in some way to the classical trial-and-error debugging techniques as we have no way of knowing what the clusters represent.

Ultimately, I believe that the benefits of iteratively constructing a causal graph outweigh the costs: it is an entity which can effectively encode human knowledge about causality in a system or application. By doing this, it allows developers or DevOps teams to discover, debug and understand problems without being experts in the subsystems or applications where these occur. It also provides a solid ground for forming hypothesis about the way a system behaves under certain situations: how would applications respond in the presence of increased queueing times? what components are affected by the network going down? The causal graph could be considered as the equivalent of a system diagram, but one which allows software to understand what points are available for measurement and how those are related to each other.

Extension of the methodology to a wider class of applications As it stands, the various steps involved in the presented methodology have been applied to I/O operations, http requests when a server is running in virtualized environments and for inferring the non-liniar functions relating variables in the causal graph [63][20]. However, more research will be required to fully understand the limitations of this approach in more complex scenarios (for example, when it is difficult to predict co-located workloads and their side-effects with respect to metrics such as available memory bandwidth).

Adoption in cloud environments While Soroban is currently mainly a research project (with the training and prediction scripts not integrated into the Resourceful API), its not unreasonable to believe that a similar tool could be deployed in cloud environments today, for helping with debugging and optimisation tasks. In the way I have described the framework here (and considering the particular example in the evaluation section), the main obstacle would be the fact that it requires changes to the application (due to Resourceful) and sometimes to the runtime environment. For example, we[21] have modified the Xen hypervisor in order to obtain domain (VM[22]) scheduling data.

[20]This is an investigation that is not part of this thesis and has been led by Philipp Geiger

[21]As mentioned previously, modifying the Xen hypervisor was in majority Oliver R.A. Chick 's contribution, with me focusing on optimisation issues.

[22]domain is the Xen term for a VM

Those type of changes are not offered today by cloud providers, and it is uncertain whether they would like to add such functionality to their hypervisors. However, it is worth noting that, besides promoting transparency, the sharing of scheduling data as we propose it does not imply the leakage of information about what other collocated VMs are executing in the cloud data-centers: each VM will only have access to its own xen-schedule-in/xen-schedule-out operations, together with metadata identifying their cause (block, yield, time slice expired, low credit etc.). For example, `lighttpd` inside a VM can not know whether it executes on the same physical host as another VM running complex database querying tasks.

The only thing that Soroban promotes is the ability of the virtualized application (i.e. `lighttpd`) to understand that sometimes variations in performance are not caused by poor local optimisations but by interference from other VMs on the same physical host. Of course, this raises the question of whether a malicious application could execute various operations (IO, CPU-intensive, Memory-bandwidth intensive) and by determining how each of those are slowed down or not, build a workload profile of other things executing on the same physical host. However, such inferences can only be done in the aggregate: the view of every single VM remains partitioned between "local" and "everything else" on the same physical host.

4.6 Conclusions

In this chapter I have presented Soroban, a potential approach for attributing performance variations at the level of application activities. In this context, attribution means the ability to discern how various components of the system or application have impacted the final measured values, and by how much. This is of high importance for anyone looking to understand unexpected application performance variations (diagnosis) and to define proper targets for optimisation (do I need to further optimise my application, or should I pay more for running it on dedicated machines in the cloud?).

For reaching the stated goal, I have proposed an attribution methodology: I suggest starting from building a causal graph that relates the measurements taken in various parts of the system. Then, based on the target variable for which attribution is desired, I have defined a process for selecting other variables which are of most benefit for attribution. Following this, a ground-truth metric must be established, that is able to assign fractions of the target variable to various components in the system. Based on the information above, a supervised learning algorithm based on Gaussian processes builds a regression model that when given the selected variables predicts the ground-truth metric for individual application activities.

The example chosen concretely was a web server (`lighttpd`) in a virtualized environment. I have shown how to apply the methodology developed in Soroban in order to attribute variations in latency either to the application or to the hypervisor. The final result was the ability to predict, for each request that was similar to at least one other request in the training data (in terms of resources consumed in the kernel), how much of the latency is given by contention at the hypervisor level.

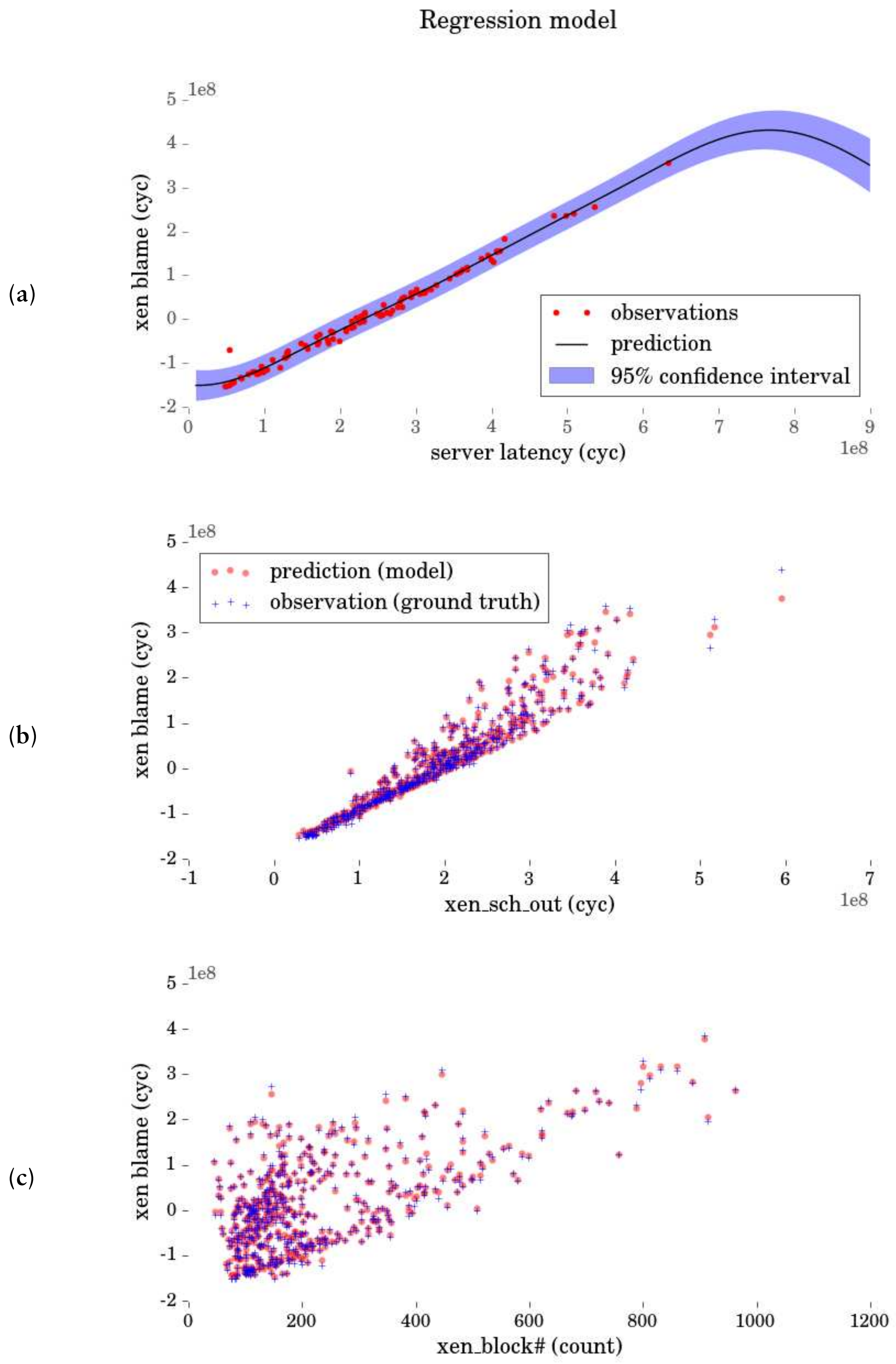


Figure 4.14: Slices through the regression model. Panel (a) considers predictions for unmeasured values (a linear range over `srv_lat`) in order to plot the prediction as a curve, together with the confidence interval Only 20% of the training observations are plotted in order to avoid cluttering the graph. Other panels simply present the predicted blame for measurements in the validation dataset.

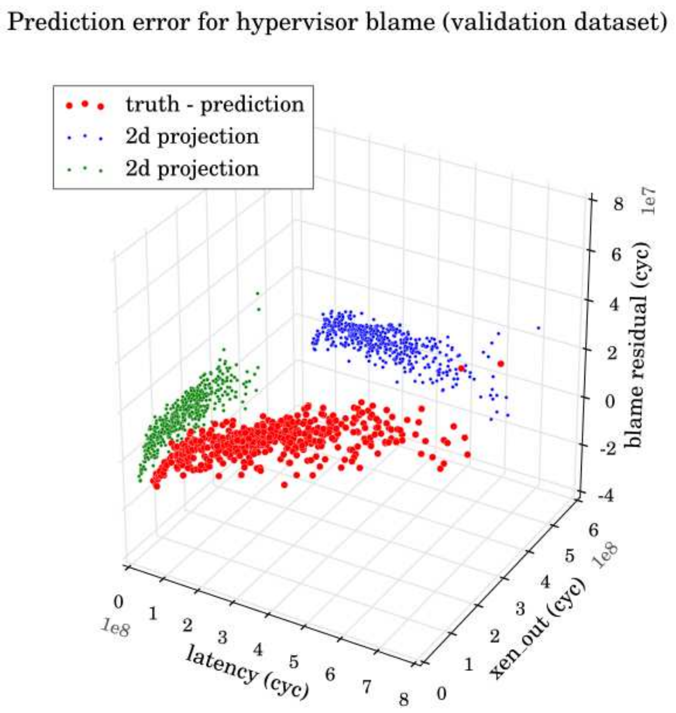


Figure 4.15: This graph shows the accuracy of the hypervisor blame prediction produced after the training process, by plotting the residuals between the model prediction and the ground truth. Only 2 out of the 5 training dimensions are represented. Note the 2D projections in green and blue and the fact that they are mostly flat (no bias in prediction power depending on variable changes) and that they are approximatively symmetric with respect to zero

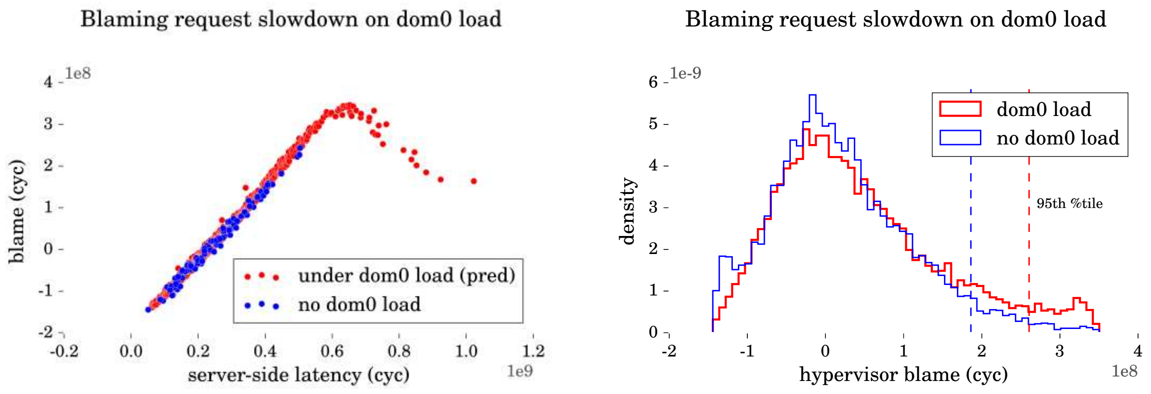


Figure 4.16: Two views detailing the predicted increase in hypervisor blame when a light load is applied to dom0 (`stress -c 1 -i 1 -d 1` : 1 cpu worker, 1 I/O worker (fsync), 1 disk worker). 15 concurrent VMs are running alongside the `lighttpd` VM, with the same workload as before. The latency increase that can be attributed to dom0 at the 95th percentile is 21ms (which represents 13 % out of the 95th percentile latency of 156ms)

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."
— Douglas Adams, The Long Dark Tea-Time of the Soul

5 Conclusion and research directions

IN THIS THESIS I have looked at using computational provenance (that is, provenance about the *properties of computations* rather than about the data they produce) as a fundamental primitive for explaining application behaviour. The main purpose in doing so was to show that this kind of metadata can be efficiently used in diagnosing unexpected variations in performance. Looking at measurement data *as a type of provenance* has allowed an understanding of the fact that kernel-level measurements need to also be related to application-level activities and that by doing so one can gain a deeper understanding of variations in performance for things which matter at the application level (requests, queries, computations, etc).

So far, this problem has not been explored from a provenance-based perspective, with most of the focus being towards ad-hoc debugging, monitoring and root-cause diagnosis tools. I believe that reconciling research done in the areas of provenance and causal statistics with work done for system measurement and diagnosis is helpful in that:

- it enables formalisations about the context of measurement; in other words, it provides the mechanisms for making sure that a measured value represents what the developer thinks it means. Surprisingly, existing strategies for doing system measurements only provide reasonable recordings for aggregated metrics such as overall load, network traffic and IO. Moving towards finer grained, semantically meaningful measurements such as "resources consumed by servicing a user request" requires the provenance-based techniques developed in Chapter 3;

- based on the above, developers and system architects could get a better idea on whether two values are *comparable*. For example, the provenance of data resulting from a benchmark will tell us whether it is reasonable to compare its results with the ones obtained from a different benchmark of a system implementing similar functionality. Furthermore, answering "what-if" questions enables the *estimation* of benchmark results under conditions closer to the ones in another benchmark, making the two comparable (in an "apples-to-apples" sense).

- it enables building semi-automated debugging tools, eliminating some of the trial and error process in identifying system issues. Expert knowledge is still helpful, but this knowledge can be encoded in ways in which both an automated system and non-experts can make use of it (causal graphs, Chapter 4). This thesis takes the first steps in that direction, showing the feasibility of such tools being built. However, more complex evaluations in realistic environments are required in order to further refine the proposed approach.

- it is compositional in nature (more speculative, not explored in detail in this thesis): two or more causal graphs dealing in detail with various parts of a complex system could be composed in order to understand the relationships between different variables in the system as a whole. There is therefore opportunity for specialisation: developers can model *local* relationships between variables measured in applications, on the kernel side or inside a hypervisor separately. A second possibility is looking at the graph at multiple “scales” (i.e.) levels of detail. This would imply nodes that have an internal structure (of further causal sub-graphs), akin to the way workflow graphs are built.

Some of the directions highlighted above remain the target of future research. However, the argument, techniques and software tools developed in the thesis show that the path is worth considering. Targeting diagnosis and attribution also had the goal of showing that provenance can play an active role and be immediately useful if considered a primitive collected during every computation. This is opposed to views of “provenance as insurance” (collecting provenance with the purpose of using it at an unspecified later time, if problems arise for a given computational result). Instead, I propose a view of provenance as metadata that allows continuous improvement, constraint checking and control of our digital infrastructure. In this context, provenance becomes part of the software we write and is then fed as input to algorithms that execute computations on it in order to extract useful pieces of information.

Chapter 3 materializes this view by describing a user-space API that applications can use for discovering the resources consumed at system level for each of the actions they perform. The actual measurements are performed by the Resourceful framework, instrumenting the relevant kernel code parts and providing back resource consumption data for each application-defined activity. The concrete examples in this thesis show how such measurements can be used for understanding performance variation, but other use cases are equally possible (e.g. resource accounting). Linking an activity to its consequences and being able to trace forward and backward between cause and effect is what makes this a form of computational provenance.

A specialised Linux kernel probing mechanism had to be developed in order to support the fine granularity at which measurements are made, and I have introduced the notion of activity tokens for mapping between kernel and user space activities. I have shown that those, together with kernel-side aggregations allow keeping time overheads to a minimum.

Future work: While the basic functionality offered through Resourceful is fairly complete, two areas for further development exist: firstly, offering the possibility of configuring measurement points and capture variables in user-space. I envision a configuration language much like the one in Listing 5.1

```
global {
  subsystem_whitelist: net_link_layer
}

subsystem net_link_layer {
  boundary:
    probe dev_queue_xmit {
      arg     : skb
      capture: {
        name: net_buf_enq,
        val : &skb->dev->qdisc
      }
    },
14      probe qdisk_restart {
15        arg    : dev
16        capture: {
17          name: net_buf_deq,
18          val : &dev->qdisc
19        }
20      }
21    metrics: cycles
22    map_async: match(net_buf_deq, net_buf_enq)
23  }
```

Listing 5.1: Sample configuration file defining a custom subsystem

This defines a custom subsystem boundary (`net_link_layer`) on two kernel functions (the probes at line 7 and 14). Furthermore, it would enable a user to set up both predefined metrics that can be captured (i.e. "cycles" on line 21) and custom asynchronous tracking. For tracking data structures, the developer would need to identify in each function a number of local variables that need to be captured (lines 9 and 16), and define a match clause with two parameters: the first declaring when data leaves a shared buffer and the second for when data is placed in that buffer.

On the Resourceful side, a hash table indexed by pointers to the captured data structures is queried whenever the custom probes are fired, and the required data for asynchronous tracking is collected.

The second possible direction for improving Resourceful would be to allow cross-host or cross-network nodes aggregations: if a given service activity crosses multiple machines, we might want to aggregate the resources consumed on each of those machines in a single coherent view. Similarly, at the network level we might want to aggregate resources consumed for particular flows. Some of the techniques for doing this have been developed for applications that base their communication on fixed specialised RPC libraries (such as Google's Protocol Buffers RPC layer) [107]. However, extending it to arbitrary applications is more challenging.

Using the data provided by Resourceful and an augmented Xen hypervisor, **Chapter 4** brings everything together into a methodology for doing attribution. Furthermore, it describes how one might evaluate the impact of changes in the configuration or in the computational environment on an executing application.

Future work: While I have provided a number of examples showing that this goal is achievable, this can only be classified as a starting point: there are numerous improvements that are needed in future research, both on the theory side (optimising the amount of required measurements) and on the inferencing side (extending structural causal models). In terms of practical implementation, Soroban needs to be integrated with visual exploration tools that allow for high level system analysis and optimisation.

Beyond the actual use cases explored in this thesis, provenance needs to be considered a primitive of computation more widely. In the next sections, I will describe some of the areas where research effort may help in supporting this vision. The list is a combination of possible provenance-based tools and longer-term desirable research.

5.1 General provenance APIs[1]

Beyond specialised APIs for provenance-enabled measurements (like the Resourceful API), it makes sense for applications to *disclose* provenance in general ways. While this section focuses

[1]An earlier version of those ideas was published as part of my article in TaPP 2015 [Lucian-2]

on what primitives are needed for making such types of disclosure possible, the end-goal needs to be at least partially automated augmentation of programs. This means that future research in static analysis might consider the needs of a provenance API in order to enable disclosure calls to be injected into either the code of applications or directly into the binary at runtime.

However, at least in existing solutions, disclosing provenance trades off the transparency of provenance capture for the possibility of recording it in a more semantically accurate way, across various layers. Workflow management systems are particularly suitable for this approach, and implementations like Kepler [4] or VisTrails [26] take advantage of the data flow and dependencies disclosed in the workflow's definition to automatically determine provenance.

Extending this to the general case assumes APIs with calls for disclosing relationships between pieces of data. Such APIs have already been proposed, either as part of observed-provenance systems, as is the case of DPAPI [102], or as general purpose provenance libraries - the case of CPL [89]. I aim to systematically discuss the issues that limit the generality of those APIs, and propose some other possible research directions.

Current API limitations: There are four major scenarios for which current available APIs fail to provide adequate support.

1. Tracking provenance at *granularities smaller than file level*. The existing solutions (both DPAPI and CPL) are able to create arbitrary provenance objects and annotate them with key/value pairs as the process transforms data. However, the main challenge is recovering the identity of those objects starting from output data. It is not currently possible to search for the provenance of some values in a file, as there is no way to determine which provenance objects hold the corresponding information. The same problem appears for operations that do not directly interact with files, but still generate provenance (as is the case with copy-pasting text between two editors).

2. Exploring the *semantics* of disclosed provenance. Existing APIs fail to consider how the key/value annotations can be consumed in automated ways (for example, by applications using provenance to reason about data quality). As long as the meaning of each key/value remains opaque, it is difficult to build applications that use provenance irrespective of its source.

3. Use in a *distributed environment*. One of the common aspects of existing provenance capturing systems is their orientation towards centralised storage. However, it would be useful to store provenance close to the data and easily keep them together when transferring between hosts or responsibility domains. Repositories similar to the ones used by distributed versioning systems (like git) would be more appropriate in this scenario, replacing huge provenance databases with structures that are more easily synchronised and managed.

4. Leveraging *existing data* as provenance. It is important to recognise that many applications already output information which could be considered provenance (e.g as part of logs or standard IO). Current provenance APIs cannot use this information directly, forcing the developer to disclose it twice when making the application provenance-aware (once as part of normal application output, and subsequently when creating provenance objects). Recognising existing data as provenance would enable applications to play an active role within a provenance-aware system without having to be modified or recompiled.

Classifying Provenance-aware Entities

Before looking at the core of a possible API, it is important to get a better understanding of the fundamental entities that have a role in generating, accumulating or propagating provenance metadata. Existing data provenance models (such as PROV-DM, OPM or Provenir) address this at a high-level, making it difficult to clarify the relationships between interacting system entities (processes, files, pipes, etc) and the provenance they produce. Instead, I consider a lower-level model, which allows a direct mapping between system object types and our API data structures.

The provenance structure and properties for different entities will vary depending on their type. For example, the provenance of some entities needs to take into account versioning (e.g. files), but this is not necessary in other cases (processes, pipes) or might be requested on demand (data structures). Existing APIs prefer a uniform approach instead.

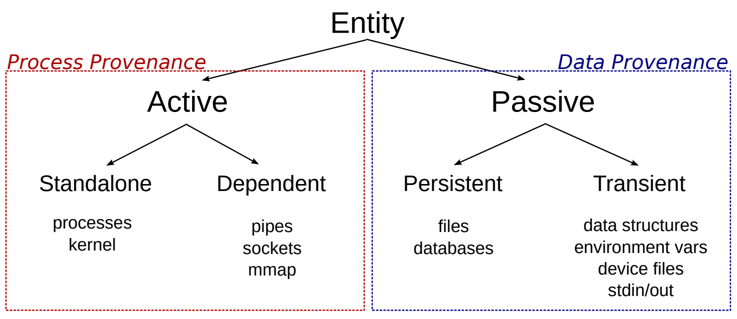


Figure 5.1: Provenance-aware entity types

As illustrated in Figure 5.1, a distinction is first made between *active* and *passive* entities, based on a simple criteria: active objects are the ones through which data “flows”, while passive objects are stores of information. This can be directly mapped on the existing classification of process provenance (active) and data provenance (passive). However, in order to obtain a complete typing of provenance, we need to distinguish subcategories within active and passive entities.

Among **active entities**, the *standalone* ones are computational: they can be instantiated from a passive entity (the binary) to produce, derive or transform data. This includes processes and OS kernels, but could also refer to computational abstractions such as middleware services.

The provenance of such entities is dual: on one hand, it is linked to the provenance of the underlying passive entity, which describes the process used to obtain the executable (the build process, compiler parameters, etc). On the other hand, standalone entities have provenance related to each particular run (command line arguments, environment variables), and are uniquely identifiable during their lifetimes.

In the other subcategory, *dependent* entities are only instantiated within the remit of another active standalone entity. They typically represent communication primitives such as pipes, sockets or memory mappings.

Passive entities map data storage abstractions and are categorised depending on how they change. The *persistent* ones are accessed or modified through fixed system interfaces (read, write) and store data for longer periods of time. *Transient entities* on the other hand have limited lifetimes, and might change without the knowledge of the OS. Typically, they live in volatile memory, even though sometimes they might be presented to the end-user as files.

We are now in the position to give a high level description of how an API should consider the accumulation and propagation of provenance: When standalone entities are instantiated, they will map the data from various passive objects (inputs, context) into local data structures (transient entities). They will then proceed to apply transformations, create new data structures, or instantiate other active objects as helpers (for further processing or communication). As dictated by internal control flow, the standalone entity will then map the results back into passive objects (files, standard output, etc). Provenance needs to track the hierarchy of active objects and the two mappings (input$\rightarrow$transient entities and transient entities$\rightarrow$output).

API design

Consider the API packaged as a library to which applications link either statically or dynamically. For minimum functionality, the application developer would just need to include one header file. At runtime, the library self-initialises, overriding part of the program startup sequence. In the process, a number of provenance objects corresponding to the standalone entity that linked to the library are automatically generated, storing information about the running process, its parent and the active context (command line arguments, environment). This means that even with minimal application changes, tracking basic process provenance would still possible. As in existing APIs, arbitrary key/value pair annotations should be added to provenance objects when required.

For data provenance, the developer needs to explicitly create those provenance objects representing passive entities, and then disclose the relationships between them and other provenance objects by calling either an `obj_relation` or the `key_relation` function. The first one discloses actual data flow between two objects (as is the case with basic input-output relationships), while the second enables associations between different key/value pairs (playing the role of a foreign key relationship).

The `key_relation` function also allows for higher levels of provenance abstraction (provenance of provenance). In this context, higher order provenance should be seen as a way of explaining existing provenance relationships. Take the example of an application that reads the name of its input file from a configuration file. First-order provenance will identify a link between the application and two particular input files. Second order provenance can explain the relationship more abstractly: namely, that the name of the second input depends on a value read from the configuration file.

Even with the functionality described so far, the sub-file granularity issue discussed in the context of existing systems is not completely solved. In order to determine the identity of a provenance object that is linked to *parts* of file, one needs to define correspondences between fragments of a passive object and their provenance (for example, expressing the fact that the first half of the file was produced by reading and processing two other files).

The `map` function implements this functionality, allowing developers to link provenance objects to specific locations within passive entities. Simple mappings could be defined between continuous (possibly overlapping) regions `[start position, end position]` and another entity. More advanced mappings would look like bit masks; automatically constructing those bit masks is possible based on explicit declarations of the output data format for a given process.

Provenance Repositories: All resulting metadata should be persisted in decentralised provenance repositories, grouping data and its associated provenance as a single manageable unit. The provenance objects which are not directly linked to any persistent entity (like the provenance of data passing through a pipe) are stored in the location of the active entity that produced them. Also, provenance from one repository can reference objects from other repositories – a key aspect of being able to scale such a system across multiple hosts.

Similar to versioning systems, provenance repositories should be managed using a dedicated tool (similar to `git`). Its purpose would be to maintain correct provenance when entities are relocated (moved locally, transferred to other systems, etc).

Fundamental limitations

Lacking static analysis tools that patch binaries with calls to a provenance API, the main limitation of any provenance API is the reliance on the correctness of developer-disclosed information. This is a problem when using provenance for security related tasks, such as intrusion detection: a virus might choose to disclose false provenance to cover its tracks and make it impossible to determine which parts of the system it has affected.

One way to overcome this would be to combine disclosed provenance with a low-overhead observed provenance system, and check for provenance consistency between the two. A trust-based model that classifies active provenance entities (such as processes) could also be a viable solution, but would require more user input.

5.2 Other directions and applications for computational provenance

Predicting the effects of local changes in distributed systems (distributed what-if scenarios) This is an extension of Soroban using data from distributed Resourceful measurements.

Dynamic instrumentation paths based on provenance: Collecting detailed instrumentation when execution diverges from a known model. This would provide a way of further reducing typical provenance collection overheads: At first, applications can be run with all available probes enabled, collecting detailed provenance data. Afterwards, instead of doing detailed instrumentation, we switch to statistical sampling in order to detect whether execution has diverged from known traces. If it has, we start detailed recording only for the "branch" that diverged. We save the new information for the subsequent runs of the same set of applications.—

Monitoring fault tolerance using provenance (infer degraded system states based on execution data paths) - this would involve tracking failure modes and measuring the resilience of measurements in the face of nondeterministic events (failure, random i/o, scheduling). A model of system execution can be built using provenance. Assuming we run the software for a long time (i.e. a server), we will get a fairly accurate image about its behavior by looking at the provenance metadata. In that case, we can identify the places where to insert instrumentation/monitor provenance for new inputs in order to see if the system has entered a state where even if it produces the correct answer, its resilience to errors is reduced. For example, in a replication system, we typically see the data replicated on 3 machines. At some point, the provenance trace shows only 2-way replication because of a storage failure. Determine that something is wrong even if the end result is correct.

Pattern discovery through provenance - similar to the clustering of code from student homework assignments proposed by Jonathan Huang[2] This was an interesting approach to determining similarities between multiple traces and grouping them together based on common properties. Clustering provenance traces might help in identifying common application patterns/problems.

Root cause analysis using "why-not" queries on provenance graphs. The study of negation in provenance graphs remains an important topic. Consider the questions I have previously asked in Chapter 4. Those can also be asked in negative form: why was a result **not** affected by a particular subsystem/application change (this is a harder problem, related to the problem of negation in logical programming languages)

Visual tools for exploring computational provenance will be fundamental in making this type of metadata accessible and in allowing interactive high-level system control. In this area, I have already developed a working prototype called Latency explorer (Figure 5.2). Here, two views of the system are made available: a latency distribution and a per-request resource consumption breakdown based on Resourceful data. Each of the parallel axes in the bottom graph identifies a consumed resource or metric specific to the application activity (here, responding to a http request). A given request is thus represented on the graph as a line linking the corresponding measured values (the red line in Figure 5.2). An idea of visual analysis using this data is to allow the selection of different intervals in the latency histogram

[2] `http://jonathan-huang.org/research/pubs/moocshop13/codeweb.html`

while coloring the corresponding requests differently in the resource consumption graph (Figure 5.3). This allows developers to understand what is different between high-latency requests and low-latency requests. Further filtering is available on each of the resource axes. Of course, tools like these could be extended to allow complex filtering and queries based

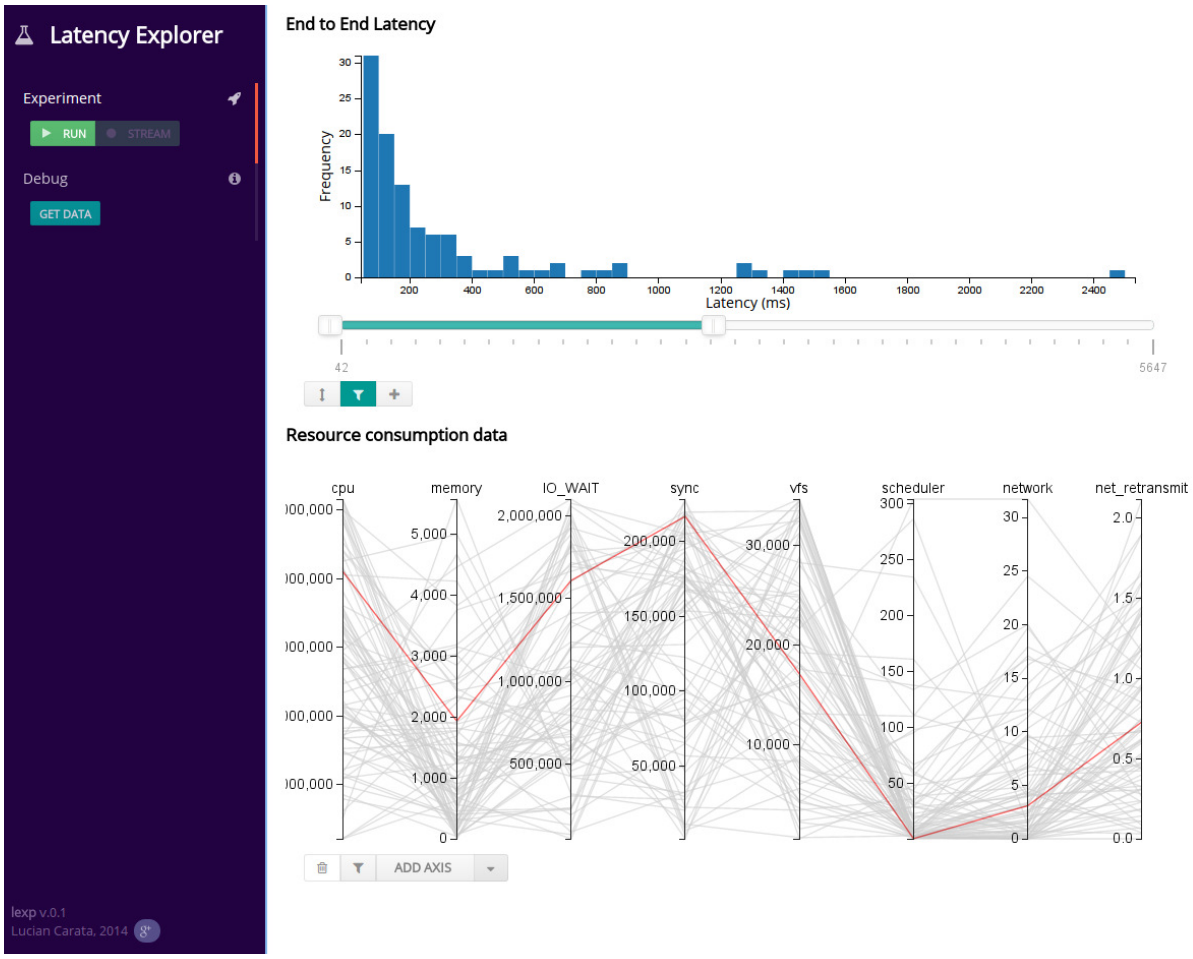


Figure 5.2: Latency Explorer, a visual analysis tool prototype

on the causal graph (surfacing Soroban results in a similarly interactive interface).

Theoretical improvements to structural causal models: The statistical theory developed for those models is not complete or adapted to the requirements of system measurement (especially considering empirical distributions and regression models instead of known functions between certain random variables) [62].

5.3 Directions and applications for data provenance

Using provenance for knowledge discovery in graphs (what path did one take to find the answer to a particular question? can this help others find information faster?) Imagine a typical browsing scenario, of searching for a particular answer and moving from one web site to the next in order to find it. The user stops when enough information was collected to answer the question. This leaves a "trail" behind (what information was used from each page and how did that contribute to answering the question). Given this trace, can other users with similar questions be guided towards the answer? (irrelevant information filtering)

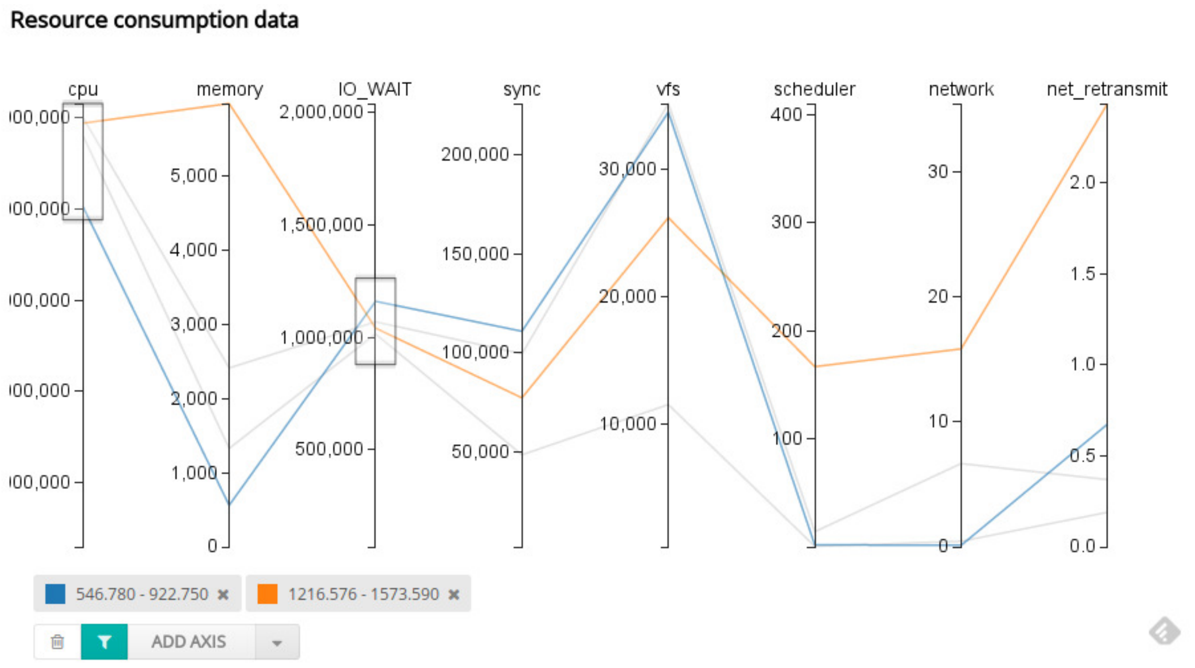


Figure 5.3: Latency Explorer, interactive filtering for comparing low-latency requests with high-latency ones. Each axis can be further filtered by the user. Here, filters are shown on the cpu and `IO_WAIT` axes.

Conditions of truth Determine the conditions/context for which a result or numerical value is true. (academic paper claims, internet-published data) Consider the provenance trace as the list of conditions necessary for the result to be true. If different provenance traces lead to the same actual result, take all into account. Now when presented with a new set of parameters/experiment conditions, we can determine on what results can we base our work. For example, you can't base your design choices on the fact that the median latency for a key-value store is 10ns if it was tested only on a get-heavy workload when your actual workload is set-heavy.

Conflicting information Making sense of data in the presence of conflicting pieces of information. If two pieces of data contradict each other, trace back the differences to changes in their respective provenance traces. What was the first place of divergence? Alternatively, if the provenance traces are the same, you know the difference is because of nondeterminism (or an error in the provenance system itself)

Measuring the quality of data by allowing user-defined (distributed) computations on top of provenance graphs. Users provide local measures of quality and aggregation operations; Overall quality can be computed based on recursively applying the local measures to the provenance data graph.

Provenance distance Determine the ways in which two pieces of data might be related (i.e the "provenance distance", akin to the "edit distance") This would allow discovery of others doing similar work (for example), or for determining what pieces of the parameter space were not explored in an experiment

Automated provenance "map" of research results in a field based on publications/running systems. Given a representation of provenance for current results in a given area (say, from papers that either publish provenance or contain sufficient information for it to be inferred) one can build a map of existing results and their lineage; By using such a map one can determine what paths were explored and what constitutes the boundary of knowledge in that field. The main way in which such a map can be used is in identifying so called "myths": results or pieces of information that keep being reused but that have since been invalidated by other research.

Merging partial data views based on provenance / Validating source independence claims (reconciling different viewpoints of a "truth") Detect self-reinforcing "myths". For example, a group of papers claiming similar but wrong results which were all based on a common bad research "seed". This is related to the classical example of journalistic "3 independent sources" when in fact all 3 have a hidden, unreliable common source.

Recording negative results in research Positive/noteworthy results are treated preferentially in academia because they generate papers. What if we could use provenance to automatically record and organise the failed attempts? One could go and see if a proposed processing path was tried unsuccessfully before committing time and resources to it

Propagating changes and invalidating results Propagate changes in personal details (home address/age/position/etc) to whoever is using this data and has permission to access it. The same techniques could be used in propagating changes from new experimental results into a larger system. This is not necessarily the same as incremental computation. I might just want to know what things were invalidated by a new discovery/result. What experiments should be re-run given a new result / or given a set of changes in my program? Based on this data, somebody may choose (for example) to do re-computation only when a provenance-based algorithm will predict that 5% of the outputs will be significantly affected. This might require weighting the contribution of each input to the outputs.

5.4 Provenance research in other areas

More generally, the research done in other fields should consider provenance as a first-class primitive with distinct characteristics.

The area of reproducible recording and execution replay [47] has so far looked at characterising software outputs while not considering computational provenance (adjacent properties of those outputs). Similarly, work is being done towards obtaining reproducible builds: the output of this research will be useful in eliminating sources of nondeterminism in the provenance of applications or libraries.

Querying and visualisation: despite the research done so far in terms of querying, exploring and visualising provenance, it still is a challenging problem and it remains to be seen how existing knowledge about graph exploration and visualisations could be applied, or whether totally different representations are required.

Distributed systems: there have been attempts of extending provenance to networked systems, but problems related to heterogeneity in distributed systems (where not all nodes are provenance-aware), scaling to a large number of nodes, long-term collection and storage remain to be solved. Additionally, there has been very little research on the provenance of the network itself (with various nodes and routes), although it would be useful for people doing work in the area of software defined networking (SDN) and virtual network functions (VNFs).

Security and privacy: we know that collecting provenance has multiple implications regarding data security and privacy, but more research is needed to understand how applications might enable provenance questions like "who is using this data?" in untrusted environments.

There is an implicit tension between provenance correctness and security: if part of the provenance graph is made inaccessible to certain parties, then those will no longer be able to run computations that extract information from the metadata or trust that the data is coming from certain original sources; for computational provenance, the existence of redacted portions in the graph allow for unknown relationships and hidden confounders to exist, which might lower the quality of any analysis starting from the underlying data. Understanding how much information is made inaccessible by securing part of the provenance graph is an area where research is certainly needed.

A possible alternate solution to this issue is introducing authorities (digital notaries) running on a model similar to certificate authorities (CAs) that would be able to certify the existence of certain relations in the provenance graph even when part of it is inaccessible.

Coming back to the initial perspective described in this thesis, provenance needs to become the underlying primitive that allows complex infrastructures to understand and take corrective action by altering their own behaviour, correcting errors and maintaining good service levels. At the same time, it should provide for the ideal level of abstraction for humans to understand and exercise control over the behaviour of dynamic algorithms such as the ones used in machine learning. The current thesis should only be seen as a first step in that direction.

Bibliography

[1] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. “Cloud monitoring: A survey”. In: *Computer Networks* 57.9 (2013), pp. 2093–2115 (cit. on p. 32).

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. “Performance Debugging for Distributed Systems of Black Boxes”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pp. 74–89 (cit. on p. 35).

[3] Bogdan Alexe, Laura Chiticariu, and Wang C. Tan. “SPIDER: a schema mapPIng DEbuggeR”. In: *VLDB ’06: Proceedings of the 32nd international conference on Very large data bases*. Seoul, Korea: VLDB Endowment, 2006, pp. 1179–1182.

[4] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-frank. “Provenance collection support in the Kepler scientific workflow system”. In: *In Proceedings of the International Provenance and Annotation Workshop (IPAW)*. Springer-Verlag, 2006, pp. 118–132 (cit. on p. 112).

[5] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. “Putting lipstick on pig: enabling database-style workflow provenance”. In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 346–357 (cit. on pp. 20, 26, 27).

[6] Mona Attariyan, Michael Chow, and Jason Flinn. “X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 307–320 (cit. on p. 34).

[7] Nikilesh Balakrishnan, Thomas Bytheway, Lucian Carata, Ripduman Sohan, and Andy Hopper. “Towards secure user-space provenance capture”. In: *Proceedings of the 8th conference on Theory and practice of provenance*. TAPP’16. 2016 (cit. on p. 24).

[8] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. “Resource Containers: A New Facility for Resource Management in Server Systems”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 45–58 (cit. on pp. 42, 81).

[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. “Using Magpie for Request Extraction and Workload Modelling”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, pp. 18–18 (cit. on p. 34).

[10] Luiz Andre Barroso. “Warehouse-Scale Computing: Entering the Teenage Decade”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: ACM, 2011 (cit. on p. 39).

[11] Sean Bechhofer et al. “Why linked data is not enough for scientists”. In: *Future Generation Computer Systems* (2011).

[12] Omar Benjelloun, Anish Sarma, Alon Halevy, and Jennifer Widom. “ULDBs: databases with uncertainty and lineage”. In: *VLDB 06 Proceedings of the 32nd international conference on Very large data bases* (2006), pp. 953–964.

[13] Mohamed N Bennani and Daniel A Menasce. “Resource allocation for autonomic data centers using analytic performance models”. In: *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*. IEEE. 2005, pp. 229–240 (cit. on p. 33).

[14] Andrew R. Bernat and Barton P. Miller. “Anywhere, any-time binary instrumentation”. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. PASTE ’11. Szeged, Hungary: ACM, 2011, pp. 9–16.

[15] Olivier Biton, Sarah Cohen-Boulakia, and Susan B Davidson. “Zoom* userviews: Querying relevant provenance in workflow systems”. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 1366–1369 (cit. on pp. 19, 25, 26).

[16] Christian Bizer, Tom Heath, and Tim Berners-Lee. “Linked Data - The Story So Far”. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.3 (Mar. 2009). Ed. by T. Heath, M. Hepp, and C. Bizer, pp. 1–22.

[17] Manuel Blum. “Coin flipping by telephone: A protocol for solving impossible problems”. In: *Advances in Cryptology-A Report on CRYPTO’81* (1982) (cit. on p. 28).

[18] Jérémy Bobbio. “Reproducible builds for Debian”. In: *FOSDEM, Feb* (2014) (cit. on p. 32).

[19] Robert L. Bocchino Jr. et al. “A Type and Effect System for Deterministic Parallel Java”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 97–116 (cit. on p. 31).

[20] Michelle A. Borkin, Chelsea S. Yeh, Madelaine Boyd, Peter Macko, Krzysztof Z. Gajos, Margo I. Seltzer, and Hanspeter Pfister. “Evaluation of Filesystem Provenance Visualization Tools.” In: *IEEE Trans. Vis. Comput. Graph.* 19.12 (2013), pp. 2476–2485 (cit. on p. 25).

[21] Rajendra Bose and James Frew. “Lineage retrieval for scientific data processing: a survey”. In: *ACM Comput. Surv.* 37 (1 Mar. 2005), pp. 1–28 (cit. on p. 13).

[22] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. “An analysis of Linux scalability to many cores”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2010.

[23] Uri Braun, Simson Garfinkel, David A. Holl, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. “Issues in automatic provenance collection”. In: *In Proc. IPAW’06, volume 4145 of LNCS*. Springer, 2006, pp. 171–183 (cit. on p. 22).

[24] Uri Braun, Avraham Shinnar, and Margo Seltzer. “Securing Provenance”. In: *The 3rd USENIX Workshop on Hot Topics in Security*. USENIX HotSec. San Jose, CA: USENIX Association, July 2008, pp. 1–5 (cit. on p. 27).

[25] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. “Why and Where: A Characterization of Data Provenance.” In: *ICDT*. Vol. 1973. Lecture Notes in Computer Science. Springer, 2001, pp. 316–330 (cit. on p. 19).

[26] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. “Vistrails: Visualization meets data management”. In: *In ACM SIGMOD*. ACM Press, 2006, pp. 745–747 (cit. on p. 112).

[27] Steven P Callahan, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva, Huy T Vo, and V INC. “Towards process provenance for existing applications”. In: *Proceedings of 2nd International Provenance and Annotation Workshop (IPAW)*. 2008, pp. 120–127 (cit. on p. 23).

[28] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. “Dynamic Instrumentation of Production Systems”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’04. Boston, MA: USENIX Association, 2004, pp. 2–2 (cit. on p. 32).

[29] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. “Dynamic Instrumentation of Production Systems”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’04. Boston, MA: USENIX Association, 2004, pp. 2–2.

[30] Giuliano Casale, Stephan Kraft, and Diwakar Krishnamurthy. “A model of storage I/O performance interference in virtualized systems”. In: *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*. IEEE. 2011, pp. 34–39 (cit. on p. 34).

[31] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. “Paxos made live: an engineering perspective”. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. PODC ’07. Portland, Oregon, USA: ACM, 2007, pp. 398–407.

[32] Adriane Chapman, Barbara Blaustein, and Chris Elsaesser. “Provenance-based belief”. In: *Proceedings of the 2nd conference on Theory and practice of provenance*. TAPP’10. San Jose, California: USENIX Association, 2010, p. 11.

[33] Adriane Chapman and Arnon Rosenthal. “Provenance Needs Incentives for Everyone”. In: *Proceedings of the third USENIX workshop on the Theory and Practice of Provenance*. TaPP. 2011.

[34] Xi Chen, Lukas Rupprecht, Rasha Osman, Peter Pietzuch, William Knottenbelt, and Felipe Franciosi. “CloudScope: Diagnosing and Managing Performance Interference in Multi-Tenant Clouds”. In: *The IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. MASCOTS ’15. 2015 (cit. on pp. 33, 43, 81).

[35] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. “Comparison of the Three CPU Schedulers in Xen”. In: *SIGMETRICS Perform. Eval. Rev.* 35.2 (Sept. 2007), pp. 42–51 (cit. on p. 34).

[36] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. “Comparison of the three CPU schedulers in Xen”. In: *SIGMETRICS Performance Evaluation Review* 35.2 (2007), pp. 42–51 (cit. on p. 81).

[37] Ron C. Chiang, Jinho Hwang, H. Howie Huang, and Timothy Wood. “Matrix: Achieving Predictable Virtual Machine Performance in the Clouds”. In: *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 45–56 (cit. on p. 32).

[38] Laura Chiticariu, Wang-chiew Tan, and Gaurav Vijayvargiya. “DBNotes: A Post-It System for Relational Databases based on Provenance”. In: *in SIGMOD ’05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM Press, 2005, pp. 942–944.

[39] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. “Understanding Data Lifetime via Whole System Simulation”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM’04. San Diego, CA: USENIX Association, 2004, pp. 22–22 (cit. on p. 31).

[40] James Clause, Wanchun Li, and Alessandro Orso. “Dytan: A Generic Dynamic Taint Analysis Framework”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. London, United Kingdom: ACM, 2007, pp. 196–206 (cit. on p. 31).

[41] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. “Tracing the lineage of view data in a warehousing environment”. In: *ACM Trans. Database Syst.* 25.2 (June 2000), pp. 179–227 (cit. on p. 19).

[42] Morgan V. Cundiff. “An introduction to the Metadata Encoding and Transmission Standard (METS)”. In: *Library Hi Tech* 22.1 (2004), pp. 52–64.

[43] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, and Sarah Cohen Boulakia. “Privacy issues in scientific workflow provenance”. In: *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*. Wands ’10. Indianapolis, Indiana: ACM, 2010, 3:1–3:6.

[44] Jeff Dean. “Achieving rapid response times in large online services”. In: *Berkeley AMPLab Cloud Seminar*. 2012 (cit. on p. 39).

[45] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Commun. ACM* 56.2 (Feb. 2013), pp. 74–80 (cit. on p. 39).

[46] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. “Eidetic Systems”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, 2014, pp. 525–540 (cit. on p. 32).

[47] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. “Eidetic Systems”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, 2014, pp. 525–540 (cit. on p. 118).

[48] Fahad Dogar, Thomas Karagiannis, Hitesh Ballani, and Ant Rowstron. *Decentralized Task-Aware Scheduling for Data Center Networks*. Tech. rep. MSR-TR-2013-96. Sept. 2013.

[49] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. “Execution Replay of Multiprocessor Virtual Machines”. In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’08. Seattle, WA, USA: ACM, 2008, pp. 121–130 (cit. on p. 31).

[50] Daniel Ellard and Margo Seltzer. “NFS Tricks and Benchmarking Traps”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’03. San Antonio, Texas: USENIX Association, 2003, pp. 16–16 (cit. on p. 81).

[51] William Enck et al. “TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones”. In: *ACM Trans. Comput. Syst.* 32.2 (June 2014), 5:1–5:29.

[52] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. “Fay: Extensible Distributed Tracing from Kernels to Clusters”. In: *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Oct. 2011 (cit. on pp. 32, 33).

[53] Peter Feiner, Angela Demke Brown, and Ashvin Goel. “Comprehensive Kernel Instrumentation via Dynamic Binary Translation”. In: *SIGPLAN Not.* 47.4 (Mar. 2012), pp. 135–146 (cit. on p. 32).

[54] Brian Fields, Shai Rubin, and Rastislav Bodık. “Focusing processor policies via critical-path prediction”. In: *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE. 2001, pp. 74–85 (cit. on p. 33).

[55] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. “X-trace: A Pervasive Network Tracing Framework”. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI’07. Cambridge, MA: USENIX Association, 2007, pp. 20–20 (cit. on p. 33).

[56] Juliana Freire, David Koop, Emanuele Santos, and Claudio T. Silva. “Provenance for Computational Tasks: A Survey”. In: *Computing in Science and Engg.* 10.3 (May 2008), pp. 11–21.

[57] Juliana Freire, Claudio T Silva, Steven P Callahan, Emanuele Santos, Carlos E Scheidegger, and Huy T Vo. “Managing rapidly-evolving scientific workflows”. In: *Provenance and Annotation of Data*. Springer, 2006, pp. 10–18 (cit. on p. 26).

[58] J. Frew, D. Metzger, and P. Slaughter. “Automatic capture and reconstruction of computational provenance”. In: *Concurrency and Computation: Practice and Experience* 20.5 (2008), pp. 485–496.

[59] Carrie Gates and Matt Bishop. “One of These Records Is Not Like the Others”. In: *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance*. Berkeley, CA, USA: USENIX Association, 2011 (cit. on p. 27).

[60] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. “Replay debugging for distributed applications”. In: *Proceedings of the annual conference on USENIX ’06 Annual Technical Conference*. ATEC ’06. Boston, MA: USENIX Association, 2006, pp. 27–27.

[61] Ashish Gehani and Dawood Tariq. “SPADE: Support for provenance auditing in distributed environments”. In: *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc. 2012, pp. 101–120 (cit. on pp. 19, 26).

[62] P. Geiger, L. Carata, and B. Schoelkopf. “Causal models for debugging and control in cloud computing”. In: (2016). arXiv: `1603.01581 [cs.AI]` (cit. on p. 116).

[63] Philipp Geiger, Lucian Carata, and Bernhard Schölkopf. “Causal models for debugging and control in cloud computing”. In: *CoRR* abs/1603.01581 (2016). arXiv: `1603.01581` (cit. on p. 105).

[64] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. "Provenance semirings". In: *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. Beijing, China: ACM, 2007, pp. 31–40 (cit. on p. 19).

[65] Brendan Gregg. "Thinking Methodically About Performance". In: *Queue* 10.12 (Dec. 2012), 40:40–40:51 (cit. on p. 33).

[66] Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. "An Architecture for Provenance Systems". In: *Contract* D3.1.1 (2006), pp. 1–5.

[67] Neil J Gunther. "Guerilla Capacity Planning". In: (2006) (cit. on p. 33).

[68] Philip J Guo and Margo Seltzer. "Burrito: wrapping your lab notebook in computational infrastructure". In: *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*. USENIX Association. 2012, pp. 7–7 (cit. on pp. 20, 23, 26).

[69] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. "Enforcing Performance Isolation Across Virtual Machines in Xen". In: *Middleware 2006*. Ed. by Maarten van Steen and Michi Henning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 342–362 (cit. on p. 34).

[70] Dani Halevy and Adi Shamir. "The LSD broadcast encryption scheme". In: *Advances in Cryptology—CRYPTO 2002*. Springer, 2002, pp. 47–60 (cit. on p. 28).

[71] Brooks Hanson, Andrew Sugden, and Bruce Alberts. "Making Data Maximally Available". In: *Science* 331.6018 (2011), p. 649. eprint: `http://www.sciencemag.org/content/331/6018/649.full.pdf`.

[72] Ragib Hasan, Radu Sion, and Marianne Winslett. "The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance." In: *FAST*. Vol. 9. 2009, pp. 1–14 (cit. on pp. 20, 27).

[73] Nima Honarmand and Josep Torrellas. "Replay Debugging: Leveraging Record and Replay for Program Debugging". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 445–456 (cit. on p. 32).

[74] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices". In: *ACM Comput. Surv.* 48.3 (Dec. 2015), 39:1–39:40 (cit. on p. 32).

[75] Galen Hunt and Doug Brubacher. "Detours: Binary interception of Win32 functions". In: *3rd USENIX Windows NT Symposium*. 1999 (cit. on p. 32).

[76] Darrel C Ince, Leslie Hatton, and John Graham-Cumming. "The case for open computer programs". In: *Nature* 482.7386 (Feb. 2012), pp. 485–488.

[77] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. "Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 469–478 (cit. on p. 35).

[78] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. "Operating System Profiling via Latency Analysis". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 89–102 (cit. on p. 32).

[79] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. “An analysis of performance interference effects in virtual environments”. In: *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE. 2007, pp. 200–209 (cit. on p. 34).

[80] Samuel Kounev, Simon Spinner, and Philipp Meier. “Introduction to Queueing Petri Nets: Modeling Formalism, Tool Support and Case Studies”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE ’12. Boston, Massachusetts, USA: ACM, 2012, pp. 9–18 (cit. on p. 33).

[81] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. “Estimating mutual information”. In: *Phys. Rev. E* 69 (6 June 2004), p. 066138 (cit. on p. 91).

[82] D P Lanter. “Design of a Lineage-Based Meta-Data Base for GIS”. In: *Cartography And Geographic Information Systems* 18.4 (1991), pp. 255–261.

[83] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. “LogGC: garbage collecting audit log”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 1005–1016 (cit. on p. 21).

[84] Yi-Ching Liao and Hanno Langweg. “Cost-benefit Analysis of Kernel Tracing Systems for Forensic Readiness”. In: *Proceedings of the 2Nd International Workshop on Security and Forensics in Communication Systems*. SFCS ’14. Kyoto, Japan: ACM, 2014, pp. 25–36 (cit. on p. 32).

[85] Robert E Llaneras, JA Salinger, and Charles A Green. “Human factors issues associated with limited ability autonomous driving systems: Drivers’ allocation of visual attention to the forward roadway”. In: *Proceedings of the 7th International Driving Symposium on Human Factors in Driver Assessment, Training and Vehicle Design*. 2013, pp. 92–98.

[86] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200 (cit. on p. 32).

[87] Chi-keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *In PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, 2005, pp. 190–200 (cit. on p. 32).

[88] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, 2015, pp. 378–393 (cit. on pp. 32, 81).

[89] Peter Macko and Margo Seltzer. “A General-purpose Provenance Library”. In: *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*. TaPP’12. Boston, MA: USENIX Association, 2012, pp. 6–6 (cit. on pp. 22, 30, 112).

[90] Peter Macko and Margo Seltzer. “Provenance map orbiter: Interactive exploration of large provenance graphs”. In: *Proceedings of the 3nd conference on Theory and practice of provenance*. TAPP’11. 2011 (cit. on p. 25).

[91] Moreno Marzolla and Raffaela Mirandola. "Performance Prediction of Web Service Workflows". In: *Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications*. QoSA'07. Medford, MA: Springer-Verlag, 2007, pp. 127–144 (cit. on p. 34).

[92] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. "Towards Practical Default-On Multi-Core Record/Replay". In: *SIGPLAN Not.* 52.4 (Apr. 2017), pp. 693–708 (cit. on p. 32).

[93] Jeanna Neefe Matthews et al. "Quantifying the Performance Isolation Properties of Virtualization Systems". In: *Proceedings of the 2007 Workshop on Experimental Computer Science*. ExpCS '07. San Diego, California: ACM, 2007 (cit. on p. 34).

[94] Kirby McCoy. *VMS File System Internals (VAX - VMS Series)*. Digital Press, 1990 (cit. on p. 31).

[95] Patrick McDaniel, Kevin Butler, Steve McLaughlin, Radu Sion, Erez Zadok, and Marianne Winslett. "Towards a secure and efficient system for end-to-end provenance". In: *Proceedings of the 2nd conference on Theory and practice of provenance*. TAPP'10. San Jose, California: USENIX Association, 2010, pp. 2–2 (cit. on p. 27).

[96] Xiaozhu Meng and B Miller. "Binary code is not easy". In: *The International Symposium on Software Testing and Analysis*. ISSTA '16. 2016 (cit. on p. 32).

[97] Jennifer C. Molloy. "The Open Knowledge Foundation: Open Data Means Better Science". In: *PLoS Biol* 9.12 (Dec. 2011).

[98] Luc Moreau and Paolo Missier. *PROV-DM: The PROV Data Model*. Technical Report. World Wide Web Consortium, Apr. 2013 (cit. on p. 23).

[99] Luc Moreau et al. "The Open Provenance Model Core Specification (V1.1)". In: *Future Gener. Comput. Syst.* 27.6 (June 2011), pp. 743–756 (cit. on p. 23).

[100] Kiran-Kumar Muniswamy-Reddy, D.A. Holland, U. Braun, and M. Seltzer. "Provenance-aware storage systems". In: *Proceedings of the 2006 USENIX Annual Technical Conference*. 2006, pp. 43–56 (cit. on pp. 19, 26).

[101] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. "Provenance for the cloud". In: *Proceedings of the 8th USENIX conference on File and storage technologies*. FAST'10. San Jose, California: USENIX Association, 2010, pp. 15–14.

[102] Kiran-Kumar Muniswamy-Reddy et al. "Layering in provenance systems". In: *Proceedings of the 2009 USENIX Annual Technical Conference*. 2009 (cit. on pp. 19, 26, 27, 30, 112).

[103] Kiran Muniswamy-Reddy and David Holland. "Causality-based versioning". In: *Proccedings of the 7th conference on File and storage technologies*. FAST '09. San Francisco, California: USENIX Association, 2009, pp. 15–28.

[104] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. "Dynamic Information Flow Tracking on Multicores". In: 2008 (cit. on p. 31).

[105] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. "Deepdive: Transparently identifying and managing performance interference in virtualized environments". In: *Proceedings of the 2013 USENIX Annual Technical Conference*. EPFL-CONF-185984. 2013 (cit. on p. 34).

[106] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. “DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments”. In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 219–230 (cit. on pp. 34, 81).

[107] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. “Diagnosing Latency in Multi-Tier Black-Box Services”. In: *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011)*. Vol. 3. 2011, p. 14 (cit. on pp. 32, 34, 35, 41, 81, 111).

[108] Stefan Otte. “Version control systems”. In: *Computer Systems and Telematics* (2009) (cit. on p. 31).

[109] Gabriele Paoloni. *White paper: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Tech. rep. 324264-001. Intel Corporation, Sept. 2010, p. 37 (cit. on p. 60).

[110] Hyunjung Park, Robert Ikeda, and Jennifer Widom. “RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows”. In: *37th International Conference on Very Large Data Bases (VLDB)*. Stanford InfoLab, Aug. 2011 (cit. on pp. 20, 26).

[111] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. “Practical Whole-System Provenance Capture”. In: *Symposium on Cloud Computing (SoCC’17)*. ACM. ACM, 2017 (cit. on p. 19).

[112] Judea Pearl. “An Introduction to Causal Inference”. In: 6 (2 2010) (cit. on p. 80).

[113] Judea Pearl. “Causal inference in statistics: An overview”. In: *Statist. Surv.* 3 (2009), pp. 96–146.

[114] Zachary N. J. Peterson, Randal Burns, Giuseppe Ateniese, and Stephen Bono. “Design and Implementation of Verifiable Audit Trails for a Versioning File System”. In: *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. FAST ’07. San Jose, CA: USENIX Association, 2007, pp. 20–20.

[115] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. “Locating system problems using dynamic instrumentation”. In: *Proc. of the 2005 Ottawa Linux Symposium*. 2005, pp. 49–64.

[116] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao. “Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds”. In: *IEEE Transactions on Services Computing* 6.3 (July 2013), pp. 314–329 (cit. on p. 34).

[117] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. “Who is your neighbour: Net i/o performance interference in virtualized clouds”. In: *IEEE Transactions on Services Computing* 6.3 (2013), pp. 314–329 (cit. on p. 34).

[118] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. “Toward the deconstruction of Dyninst”. In: *Univ. of Wisconsin, technical report* (2007) (cit. on p. 32).

[119] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. “Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers”. In: *IEEE Micro* (2010), pp. 65–79.

[120] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 199–212 (cit. on p. 27).

[121] Thomas G Robertazzi. *Computer networks and systems: queueing theory and performance evaluation*. Springer Science & Business Media, 2012 (cit. on p. 33).

[122] Satya S. Sahoo and Amit Sheth. “Provenir ontology: Towards a Framework for eScience Provenance Management”. In: *Microsoft eScience Workshop*. Pittsburgh, PA, 2009.

[123] Can Sar and Pei Cao. *Lineage File System*. Tech. rep. Department of Computer Science, Stanford University, Jan. 2005.

[124] Prateek Saxena, R Sekar, and Varun Puranik. “Efficient Fine-grained Binary Instrumentationwith Applications to Taint-tracking”. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’08. Boston, MA, USA: ACM, 2008, pp. 74–83 (cit. on p. 22).

[125] Carlos Scheidegger, David Koop, Emanuele Santos, Huy Vo, Steven Callahan, Juliana Freire, and Cláudio Silva. “Tackling the Provenance Challenge one layer at a time”. In: *Concurrency and Computation: Practice and Experience* 20.5 (2008), pp. 473–483 (cit. on pp. 19, 22, 25).

[126] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. “The Seven Deadly Sins of Cloud Computing Research”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud’12. Boston, MA: USENIX Association, 2012, pp. 1–1 (cit. on p. 81).

[127] M.I. Seltzer, P. Macko, and M.A. Chiarini. “Collecting Provenance via the Xen Hypervisor”. In: *Proceedings of 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP’11), June*. 2011, pp. 20–21.

[128] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on p. 28).

[129] Weijie Shi, Linquan Zhang, Chuan Wu, Zongpeng Li, and Francis C.M. Lau. “An Online Auction Framework for Dynamic Resource Provisioning in Cloud Computing”. In: *The ACM International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’14. ACM, 2014, pp. 71–83.

[130] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. “Seawall: Performance Isolation for Cloud Datacenter Networks”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010 (cit. on p. 81).

[131] Chen Shou, Dongfang Zhao, Tanu Malik, and Ioan Raicu. “FusionProv: Towards a Provenance-Aware Distributed Filesystem”. In: *Greater Chicago Area System Research Workshop (GCASR)*. 2013 (cit. on p. 31).

[132] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010 (cit. on pp. 32, 33).

[133] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. “A survey of data provenance in e-science”. In: *SIGMOD Rec.* 34.3 (Sept. 2005), pp. 31–36.

[134] Yogesh Simmhan, Beth Plale, and Dennis Gannon. *A survey of data provenance techniques*. Technical Report TR-618. Computer Science Department, Indiana University, 2005.

[135] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. “Secure Program Execution via Dynamic Information Flow Tracking”. In: *SIGARCH Comput. Archit. News* 32.5 (Oct. 2004), pp. 85–96.

[136] Salmin Sultana and Elisa Bertino. “A File Provenance System”. In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. CODASPY ’13. San Antonio, Texas, USA: ACM, 2013, pp. 153–156 (cit. on p. 31).

[137] Ariel Tamches. “Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels”. PhD thesis. University of Wisconsin-Madison, Feb. 2001.

[138] Wang Chiew Tan. “Provenance in Databases: Past, Current, and Future”. In: *IEEE Data Eng. Bull.* 30.4 (2007), pp. 3–12.

[139] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. “Modeling Virtual Machine Performance: Challenges and Approaches”. In: *SIGMETRICS Perform. Eval. Rev.* 37.3 (Jan. 2010), pp. 55–60 (cit. on p. 34).

[140] *Tux3, a Versioning Filesystem*. `https://lkml.org/lkml/2008/7/23/257`. Accessed: 2018-04-15 (cit. on p. 31).

[141] Owen Vallis, Jordan Hochenbaum, and Arun Kejariwal. “A novel technique for long-term anomaly detection in the cloud”. In: *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. 2014 (cit. on p. 32).

[142] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. “DoublePlay: Parallelizing Sequential Logging and Replay”. In: *SIGPLAN Not.* 47.4 (Mar. 2011), pp. 15–26 (cit. on p. 31).

[143] S. Votke, S. A. Javadi, and A. Gandhi. “Modeling and Analysis of Performance Under Interference in the Cloud”. In: *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sept. 2017, pp. 232–243 (cit. on p. 34).

[144] Robert NM Watson and Wayne Salamon. “The FreeBSD audit system”. In: *UKUUG LISA Conference, Durham, UK*. 2006.

[145] Lisa Wells. “Performance analysis using coloured Petri nets”. In: *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*. IEEE. 2002, pp. 217–221 (cit. on p. 33).

[146] Jennifer Widom. *Trio: A System for Integrated Management of Data, Accuracy, and Lineage*. Technical Report 2004-40. Stanford InfoLab, Aug. 2004 (cit. on p. 19).

[147] Chadd C Williams and Jeffrey K Hollingsworth. “Interactive binary instrumentation”. In: *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*. 2004 (cit. on p. 32).

[148] Allison Woodruff and Michael Stonebraker. “Supporting Fine-grained Data Lineage in a Database Visualization Environment”. In: *Proceedings of the Thirteenth International Conference on Data Engineering*. ICDE ’97. IEEE Computer Society, 1997, pp. 91–102.

[149] Min Xu, Rastislav Bodik, and Mark D. Hill. “A ”Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay”. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA ’03. San Diego, California: ACM, 2003, pp. 122–135 (cit. on p. 31).

[150] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. “ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay”. In: *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)* (2007) (cit. on p. 31).

[151] Ziye Yang, Haifeng Fang, Yingjun Wu, Chungi Li, Bin Zhao, and H Howie Huang. “Understanding the effects of hypervisor I/O scheduling for virtual machine performance interference”. In: *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE. 2012, pp. 34–41 (cit. on p. 34).

[152] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. “Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 116–127 (cit. on p. 31).

[153] Takeshi Yoshino, Yutaka Sugawara, Katsushi Inagami, Junji Tamatsukuri, Mary Inaba, and Kei Hiraki. “Performance Optimization of TCP/IP over 10 Gigabit Ethernet by Precise Instrumentation”. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC ’08. Austin, Texas: IEEE Press, 2008, 11:1–11:12 (cit. on p. 51).