Issues Arising in Using Kernel Traces to Make a Performance Model

Murray Woodside
Carleton University
Ottawa, Canada
cmw@sce.carleton.ca

Shieryn Tjandra
Carleton University
Ottawa, Canada
shieryntjandra@gmail.com

Gabriel Seyoum
Carleton University
Ottawa, Canada
gaseyoum@gmail.com

ABSTRACT

This report is prompted by some recent experience with building performance models from kernel traces recorded by LTTng, a tracer that is part of Linux, and by observing other researchers who are analyzing performance issues directly from the traces. It briefly distinguishes the scope of the two approaches, regarding the model as an abstraction of the trace, and the model-building as a form of machine learning. For model building it then discusses how various limitations of the kernel trace information limit the model and its capabilities and how the limitations might be overcome by using additional information of different kinds. The overall perspective is a tradeoff between effort and model capability.

CCS CONCEPTS

•General and reference~Cross-computing tools and techniques~Measurement •General and reference~Cross-computing tools and techniques~Performance

KEYWORDS

Performance model; Layered queueing; Kernel traces.

ACM Reference format:

Murray Woodside, Shieryn Tjandra, Gabriel Seyoum. 2020. Issues Arising in Using Kernel Traces to Make a Performance Model. In *Proceedings WOSPC-2020, ICPE 2020 Companion, April 20-24, Edmonton, AB, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3375555.3384937

1 Introduction: Kernel Traces

Kernel traces are gathered from operating system events identified with process and thread ids, network interface ids and file ids. Together with other OS-gathered metrics they have the advantage of not requiring any application instrumentation, and gathering data for applications for which there is no available source code (or for components for which there is no source code). They are also extremely fine-grained, and include all processor-scheduling events, so everything which stops a process from running can be identified.

LTTng [9] is a tracing tool that is distributed with Linux, so it is always available on a Linux system. A large range of types of pre-defined events to be recorded can be selectively enabled, relating to CPU scheduling, signals, locks, and network interfaces [9]. It has low overhead, but it only records into a main-memory buffer which may limit the length of a tracing run. Tracing all scheduler events takes a lot of memory but is required for deep analysis. LTTng also has an API to record user-defined events in the application code, to integrate kernel and application level events in one trace.

The DORSAL group at Ecole Polytechnique de Montreal, under M. Dagenais, have contributed to LTTng and are using it to analyse performance issues with the analysis tool TraceCompass [12]. Among TraceCompass's capabilities two are most noteworthy. For distributed systems it is capable of synchronizing the clock times in traces gathered separately on different hosts (it adjusts clock offsets by applying causal ordering to messaging events that flow between the hosts). For performance analysis they have developed a "waiting analysis" [4] which traces what every process is waiting for when it is scheduled, through the sequence of scheduler events. If there is a bottleneck process or lock it shows up in all the waiting conditions. They also display a "critical path" of chains of events linked by waiting, which may traverse multiple processors. This has been extended in various ways, e.g. to diagnose performance deviations between different executions, by examining call stacks at points along the critical path [2].

2. Layered Models based on Traces

The author and co-workers developed tools to extract layered performance models from interprocess message traces [5][6]. Patterns of messages were interpreted as various kinds of process interactions, and an interaction architecture at the process level was extracted. Application level trace events were used to identify where a process sent a request and where it blocked to wait for a reply. This gave the structure of a layered model and the call frequencies between processes but the calibration of CPU time was left to separate operations using methods such as surveyed in [8]and implemented in LibReDE [11]. Model extraction went further in [7] and modeled parallel paths and multithreaded tasks, and determined the CPU times of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICPE '20 Companion, April 20–24, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7109-4/20/04?$15.00
https://doi.org/10.1145/3375555.3384937

operations. Operations were clustered and considered to be instances of the same operation, based on their interactions (i.e. based on what other processes they made requests to).

Layered models are described in [3]. They are an extension of queueing models to describe software servers as well as hardware. The software part is structured as in Fig 1, with *tasks* (= concurrent processes), *entries* (their operations) and *calls* (RPCs), and labels for the mean CPU demands of operations, and for the mean number of calls. Figure 1 shows an example with one entry per task, and blocking calls between the entries. When a program of this architecture was traced and a model extracted using the approach of [6], it was as shown in the automatically drawn Figure 2. The entry and task names were automatically generated by the analyzer. To solve the model its deployment to host processors must also be defined; the model builder did not capture these.

In Figure 2 there are additional entries, not in Figure 1, derived from operation instances in the trace that made different collections of calls. Thus in task Server02, E_2S represents one or more operation instances that made no calls, and E8S represents one or more that called both Server04 and Server05. There are also additional types of calls; in task Server01 there are two entries that receive asynchronous messages from the User task (shown with open arrowheads in this Figure).

Sometimes the extracted model has a substantial clutter of operations that were performed during initialization, for instance of the RPC. Usually this part of the operation is not critical for performance and can be ignored if an application level event can be added at the beginning of the operational part of the scenario. In tracing performance tests, such an event was added to the test driver that simulated the users.

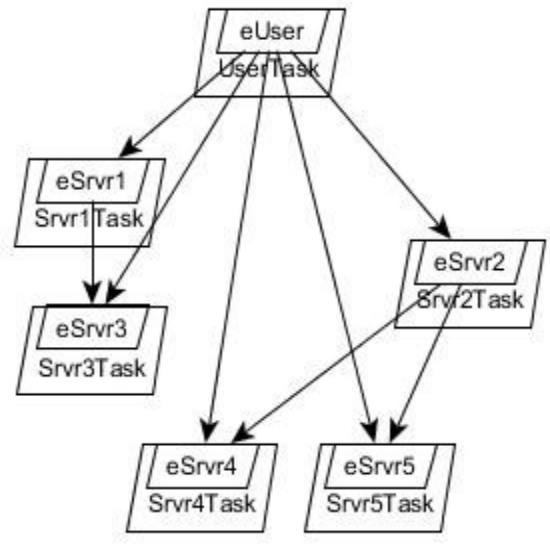

Figure 1: A layered architecture

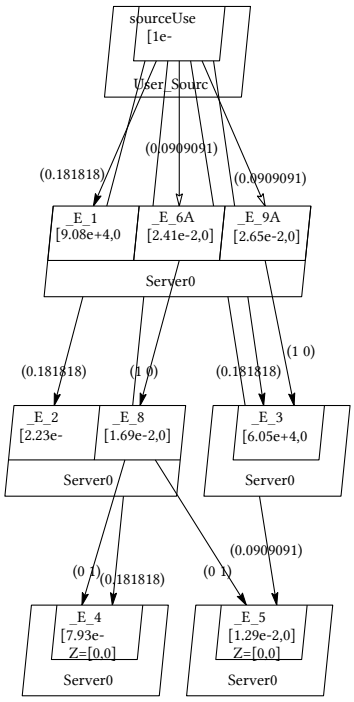

Figure 2: A layered performance model, extracted from a system, that describes the architecture in Figure 1

3. Performance Analysis from Traces

A trace can be exploited by both *direct analysis* and by *model-based analysis*. As an example of a powerful direct analysis, "wait analysis" as developed by Dagenais and his co-workers [4] can identify an operating system entity (such as a process thread, a system lock or a peripheral device) which is frequently the cause of waiting by other entities. This extends to include causal chains such as causality across a network, where some processes often wait for a message from a particular remote process. For performance analysis Dagenais and his co-workers have developed a "waiting analysis" [4] which traces what every process is waiting for when it is scheduled, through the sequence of scheduler events. If there is a bottleneck process or lock it shows up in all the waiting conditions. Effectively the wait analysis can deduce: "over the duration of this trace, this entity is a suspect". So it is useful in diagnosis, and in the hunt for places to make improvements. They also display a "critical path" of chains of events linked by waiting, which may traverse multiple processors. This has been extended in various ways, e.g. to diagnose performance deviations between different executions, by examining call stacks at points along the critical path [2].

All of these direct analyses however are limited to what has been observed over a particular time period and lack both generality and predictive capability.

Machine-learning models could also be applied, to abstract away from the trace and summarize the results as a model that can be used to interpolate in the results. They are (in most cases) like curve-fit models; they support interpolation within the observed cases but not prediction to new situations. For extrapolation, performance models are needed.

Regarding *performance model-based analysis*, a performance model is essentially an abstraction tool for extrapolating from the observations in the trace. It may extrapolate to different load levels, deployments, or to systems with additional subsystems. Extrapolation is based on assumptions (1) that the model captures the system structure and (2) that the parameters fitted to the trace remain valid when extrapolating. The second assumption can be relaxed by re-fitting the parameters using methods such as in REF. Essentially successful extrapolation rests on the model capturing general behaviour like contention effects, which applies to situations beyond the traced experiments or not.

The trace does not have to capture a performance problem to give a useful model; a light-load trace can predict a bottleneck location even though the bottleneck is not active. However a model made this way is still limited by the conditions under which the trace was made. The limitations are related to its *structure* and its *parameters*. For structure, some important entities may be missing from the model because they are not represented in the trace. For parameters, the execution of an operation depends on environment factors at the time of execution, for example on the state of caches and on the processor load (through power management and speed control).

4. Limitations of Trace-Based Models and How to Deal With Them

The following issues are examined specifically for kernel traces, although many of them also apply to application-level traces. We try to identify additional information that can improve the model usefulness for various purposes, and its associated effort.

Missing model *structure* may include;

- The identity of the program for each process id: for interpreted software such as Java or Python, the process id is associated with the interpreter program rather than the application code.

  What can be done: This is not difficult. The proc file system (a monitoring repository maintained by Linux) records the command line that invoked the process, so the application file can be discovered in that data if it is queried or recorded periodically during tracing.

- Software resources such as locks that the analyst knows nothing about, implemented in the application (e.g, a software-implemented lock or a limited buffer pool). are invisible.

  What can be done: This complicated. The existence of such a resource could be inferred from unexplained waiting in the trace data. First, suppose a process A is waiting for a software lock; it executes after waiting for another process B (which has released the lock). The causality could be a message or signal from B, if these can be eliminated (because they are detected separately as interprocess events) then the release of a logical resource is a candidate cause. Additional inferences which would be needed include some way of identifying other interactions with the same logical resource. If two logical resources are implemented in the same process they cannot be distinguished in this way.

  Second, the resource could be inferred from a separate model-building approach based on fitting the model to average performance measures. If the delays of the best fitted model are not an adequate match to the data, an additional queue not related to known queueing resources, could be hypothesized and its contribution to a better fit could be tested statistically (REF reg). This is a big additional effort, however it could be done as part of parameter fitting used to maintain the model, see below.

- The destination of messages via messaging middleware such as MQ is hidden from the sender, which only sees an interaction with the message broker. Thus the trace does not reveal what service is being requested.

  What can be done: Here it is essential to trace at least the middleware, to capture the arrival of a message and trace it through to its destination(s). If not the middleware, then the application might provide the information. Middleware tracing seems to be an essential add-on to kernel tracing, and is feasible for open-source middleware. Once installed it can be re-used in any other system. However there are many kind of middleware.

- The direction of calling in interprocess calls via messages is not knowable just from the kernel level message events. For example for RPCs the kernel simply sees a sequence of messages, with no call/reply semantics. It is natural to interpret the first message between two processes as a call, and the next in the reverse direction as the corresponding reply. However if the first message is asynchronous, the second may be a callback initiated by its receiver, in the reverse direction, reversing the sense of an entire sequence of calls.

  What can be done: In event-driven software this is not an issue, since all messaging is asynchronous anyway. Otherwise this requires some application or middleware level knowledge about which process is making interprocess calls. Simplest would be to trace the calls and replies at the application level. Some middleware supports client-server relationships and keeps track of the roles of its users; if the messages are traced at this level they can be distinguished (but MQ, for example, does not).

- Requests to remote servers that are themselves not monitored, are only partially covered. The occurrence of the request message is traced, and the reply from the remote server, but connecting the two requires message data such as the URL of the remote server, which is not accessible in the kernel trace. An important example is file operations on a storage server for applications in the cloud.

In large-scale systems remote servers are a serious issue, and there is also a larger issue of whether such servers are wanted in the model.

What can be done: First, a server can be assumed, for the remote requests. If there are several of them they may not be distinguishable. Second, if the remote servers are known, as in a test environment, it may be possible to infer which messages go to which server. Third, the content of a packet can be captured in the trace, so it might be possible to parse out the identity of the remote server, although the storage required for this might be prohibitive.

- operations by peripheral devices, that are not traced. File I/O devices are an important example. For file operations LTTng can record the syscalls to identify operations, but they apply to anything mapped to a file... sockets for instance. For a single operation there may be multiple read or write calls, each for part of the file.

  What can be done: Possibly the peripheral device was just not used during the trace, so a longer trace may capture its usage. The same potentially could apply to a process that is not activated during a trace. For file reads and writes the OS separately records total blocks read and written (but not by file), which can possibly be incorporated in the model parameters, especially if only one file is open at a time.

Also unwanted model structure (clutter) may be produced:

- As noted in the example of Figure 1/2, some execution periods (such as initialization) may have complex behaviour that is not of interest to the analyst. If events can be recorded to demarcate these periods, they can be eliminated from the modeling effort. This approach can be extended to create separate models for different phases of execution of the entire application, using boundary events related to the beginning of a new phase.

  What can be done: This requires including application level events in the trace, which can be done with LTTng. To avoid having to instrument the source code it may be possible to capture the boundaries of such a period in a custom-written component, such as the test driver mentioned above. Another potential source of boundary events is existing application logging such as a weblog.
- Multi-threaded programs require tracing the behaviour of each thread and treating it as a separate concurrent task in the model, since causality between stimulus and response flows through each thread separately. If each one is modeled as a task it may create a huge model.

  What can be done: If these threads have essentially the same behaviour, as in a server thread pool, they should be collected together as a software multiserver. If a thread is just a unit of modular behaviour, created dynamically to execute a function and then destroyed, it can be absorbed into the creating thread if that one is blocked. The analysis for these characteristics may be complex, however.
- It may be possible only to trace a subsystem at a time, in a large distributed system, because of the amount of data involved.

  What can be done: subsystem models can be connected in a compositional approach, in various modeling systems. For layered modeling a compositional approach was described by Wu et al [14].

Model parameter weaknesses can include:

- the cpu demand of operations and the number of calls to them can be calculated from the scheduling events of the trace, but may not be representative.

  What can be done: Longer traces taken under a variety of conditions may be needed to get representative averages. Also, the parameters can be calibrated separately (after creating the model structure) from performance measures which require much simpler logging and can be done over a much longer time. CPU demand calibration in this way is well developed REF.
- multi-event messages are common when a large block of data is transferred with several send syscalls. Normally in a performance model we would like to identify them as one message, with one remote operation and one delay for the reply. Clustering these send or receive events is a challenge without application level identification of the application message.

  What can be done: heuristics are only partly successful. For example one tool we use identifies the last message of a cluster of sends and the first message back as a send-reply, and all the others are identified as asynchronous messages; a heuristic is then to cluster the asynchronous messages before and after with the send-reply. However this may disregard an actual asynchronous message.

5. Conclusions

Trace interpretation can recover essential causal information such as the request message which causes a process to execute an operation, and the set of events triggered by that message. Kernel traces are valuable in that they capture all the events of the given type, without requiring user instrumentation effort. This is also the source of their weaknesses: because they are at kernel level they lack application semantics. Clever inferences or partial structural knowledge can sometimes fill these gaps, but some absolutely require additional data. A major example for modern distributed systems is monitoring of the middleware to connect the sender of a message with its receiver. Some other kinds of additional data may be readily available in some cases, such as advance knowledge of some of the interprocess interactions and of untraced remote servers. A second important example for many systems is the ability to separate out initialization activity from production activity, in order to model only the latter.

ACKNOWLEDGMENTS

This research was supported by NSERC (the Natural Sciences and Engineering Research Council of Canada) in its Discovery Grant program, grant RGPIN 06274-2016, and by a MITACS Accelerate grant IT11285.

REFERENCES

[1] M. Desnoyers, M.R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux", in OLS (Ottawa Linux Symposium), pp. 209–224, 2006.

[2] F. Doray, M. Dagenais, "Diagnosing Performance Variations by Comparing Multi-Level Execution Traces", IEEE Trans. on Parallel and Distributed Systems, v 28 n 2 pp 462-474, 2017.

[3] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks", IEEE Trans. on Software Eng. Aug. 2008

[4] F. Giraldeau, M. Dagenais, "Wait Analysis of Distributed Systems Using Kernel Tracing," IEEE Trans. on Parallel and Distributed Systems , V 27, Issue 8, pp 2450 – 2461, 2016.

[5] C. Hrischuk, J. Rolia and C.M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", Proc. MASCOTS 1995, pp. 399-409, January 1995

[6] T.A. Israr, D.H. Lau, G. Franks, M. Woodside, "Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces", Proc. 5th Int. Workshop on Software and Performance (WOSP 2005), pp 147-158, July 2005.

[7] Mizan and G. Franks, "An automatic trace based performance evaluation model building for parallel distributed systems," Proc. Joint WOSP/SIPEW Int. Conf. on Performance Engineering (ICPE 2011), Karlsruhe, pp. 61-72, April 2011.

[8] S. Spinner, G. Casale, F. Brosig, S. Kounev. "Evaluating Approaches to Resource Demand Estimation", Performance Evaluation, Elsevier 92:51-71, Oct 2015

[9] The LTTng Documentation, https://lttng.org/docs/v2.11/, accessed January 2020,

[10] S. Tjandra, "Performance Model Extraction Using Kernel Event Tracing", MASc thesis, Carleton University, 2019.

[11] S. Spinner, J, Grohmann, "LibReDE User Guide", https://se.informatik.uni-wuerzburg.de/fileadmin/10030200/user_upload/librede/LibReDE_UserGuide_01.pdf, February 28, 2019

[12] Trace Compass User Guide, https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/User-Guide.html, accessed Jan 2020

[13] M. Woodside, "The Relationship of Performance Models to Data", Proc. SPEC Int. Workshop on Performance Evaluation, Springer LNCS, Vol. 5119, pp 9 - 28, June 2008.

[14] Xiuping Wu and Murray Woodside, "Performance Modeling from Software Components," in Proc. 4th Int. Workshop on Software and Performance (WOSP 04), Jan 2004, pp. 290-301..