

Unifying Event Trace and Sampled Performance Data

Nathan Scott
Monash University, Australia

Abstract. Understanding the performance of distributed computing systems is a complex and highly specialised task. Ideally, information from all aspects of each system involved would be used to inform an analysts view of an individual component's contribution to a variety of performance criteria. This study investigates the merit of extending an existing sampling-based set of performance analysis tools to incorporate event trace data into the set of available metrics. Case studies of this approach to performance analysis in production environments has been undertaken. Initial progress suggests the richer collection of information has improved our analytic capabilities. Deeper insight can be gained from bridging the gap between traditional methods of performance data gathering and the increasingly popular event tracing paradigm can be gained.

Keywords. Performance monitoring, performance analysis, problem diagnosis, complex systems, event tracing, dynamic tracing, static tracing, sampling, event counters, distributed systems, instrumentation, visualisation, quality-of-service, performance tools, system administration.

Table of Contents

Overview	2
Review	3
<i>Magpie</i>	3
<i>DTrace</i>	4
<i>SystemTap</i>	5
<i>X-Trace</i>	5
<i>LTT, LTTng and UST</i>	7
<i>Dapper</i>	8
<i>Intrusion Detection</i>	9
<i>NetLogger</i>	9
<i>Mixing Metaphors</i>	10
<i>Performance Co-Pilot</i>	11
<i>Unified Metrics Namespace</i>	12
<i>Sampling-based Agents</i>	13
<i>Sampling-based Visualisation</i>	13
Method	13
<i>Software Artefacts</i>	14
<i>Event Trace Agents and Queueing</i>	14
<i>Visualising Event Traces</i>	15
Log Streaming Case Study	17
<i>Centralising Event Logs</i>	17
<i>Deployment and Instrumentation</i>	18
<i>Event Analysis and Filtering</i>	20
<i>Security-sensitive Parameters</i>	21
Data Warehouse Case Study	22
<i>Instrumenting the Extraction Phase</i>	23
<i>Process Visualisation</i>	24
Future Work	26
Conclusion	27
References	28

Figures and Tables

<i>Figure 1: X-Trace Node Visualisation</i>	6
<i>Figure 2: X-Trace Flow Visualisation</i>	6
<i>Figure 3: Structure of X-Trace metadata</i>	6
<i>Figure 4: Dapper Spans and Trees</i>	8
<i>Figure 5: Intrusion Detection Visualisation</i>	9
<i>Figure 6: NetLogger Lifeline Visualisation</i>	10
<i>Figure 7: Distributed PCP Components</i>	12
<i>Figure 8: Interactive Strip Charts</i>	13
<i>Figure 9: Event Trace Agent Queues</i>	14
<i>Figure 10: Unified Strip Chart Visualisation</i>	15
<i>Figure 11: Extended Trace Selection Model</i>	16
<i>Figure 12: Log Stream Design</i>	17
<i>Table 1: Configuring rsyslog</i>	18
<i>Figure 13: System Log Event Arrivals</i>	19
<i>Figure 14: System Log Event Throughput</i>	19
<i>Figure 15: Event Reporting and Filtering</i>	20
<i>Figure 16: A Performance Cube</i>	22
<i>Figure 17: Sample Data Warehouse Report</i>	22
<i>Table 2: Shell Event Trace Instrumentation</i>	23
<i>Figure 18: Sample Script Instrumentation</i>	24
<i>Figure 19: Warehouse Import Drill-down</i>	25
<i>Figure 20: Retrospective PCP Analysis</i>	26

Overview

Understanding where time has been spent in performing a computation or servicing a request is at the forefront of the performance analyst's mind. Measurements are available from every layer of a computing system, from the lowest level of the hardware up to the top of the distributed application stack. In recent years we have seen the emergence of tools which can be used to directly *trace* events relevant to performance. This is augmenting the traditional event count and system state instrumentation, and together they can provide a very detailed view of activity in the complex computing systems prevalent today.

We can categorise performance instrumentation approaches into two broad categories which we hereafter refer to as *sampled values* and *event traces*.

Sampled values are typically cumulative counters of some event, resource, quantity or time, or they are an instantaneous observation or a discrete value. These metrics are either inexpensive to update or are maintained for functional purposes independent to the need for instrumentation, are often permanently enabled and are exported from the instrumented component using statistical sampling (Jain, 1991). The instrumentation and interpretation techniques are generally simple, well researched, and widely used in the field - as a result these sampled values are ubiquitous in all modern computing systems.

Event traces record information about individual events as they happen. An event trace is an asynchronous timestamped sequence of one or more events, where the events may be milestones in an operation, such as its start or end point, or a singular observation. Data about that specific event is often part of the trace data (Barham et al., 2003) which aids understanding the nature of the event. Events can also be nested, such that one event may be associated with a parent or existing event being traced. More sophisticated tracing tools are then able to perform end-to-end tracing of events which can span multiple subsystems (Barham et al., 2004 and Fonseca et al., 2007), and can even tie together traces from completely unrelated software and hardware components, allowing construction of a timeline graph showing in each subsystem where time has been spent in servicing an individual request.

A request, in this context, should be considered in the broadest possible scope – being not limited to something as short and tightly defined as a service request (device operation, database request, filesystem request) – but also potentially any end user request (a web application interaction, for example). One request may be nested within another, and they may span thread, process and system boundaries.

Event tracing has the advantage of keeping the performance data tied to the individual requests, allowing deep inspection of a request which is useful when performance problems arise. The technique is also exceptionally well suited to exposing transient latency problems (Bligh et al. 2007). The downsides are increased overheads (sometimes significantly) in terms of instrumentation costs as well as volumes of information produced. To address this, every effort is taken to reduce the cost of tracing - it is common for tracing to be enabled only conditionally, or even dynamically inserted into the instrumented software and removed when no longer being used.

Performance analysis requires visibility into all aspects of system performance, including end user response time, throughput, latency, resource utilisation, and bottleneck identification. Generic tools that can extract performance metrics from multiple systems and from the different subsystems within each system - hardware, operating system, system services, databases, and applications - are invaluable. When using such generic tools one can much more readily apply general rules and lessons learnt about performance data and modelling from one situation to the next (Gunther, 2004).

Traditionally, we have predominantly used sampling techniques, where metric values are sampled on a relatively long (usually fixed) time interval, and over time we make observations about resource utilisation, throughput, queue lengths, and so on. Such approaches have the advantage of having low impact on the monitored system, provide an excellent starting point for diagnosing performance problems, and lend themselves to tasks like resource monitoring over long periods of time, performance modelling, and capacity planning.

The relatively long sampling intervals used, while keeping sampling costs low, sacrifice the ability to perfectly reconstruct the actual system behaviour. This would require significantly shorter sampling periods, as defined by the Nyquist-Shannon sampling theorem (Shannon, 1949). To avoid the inherent cost and associated disturbance to the system under observation, we augment our baseline sampling-based metrics with higher fidelity trace-based metrics.

Tracing technologies with low impact and high levels of safety when enabled in production environments are becoming increasingly prevalent, suggesting the information they provide is of high value in the diagnosis of performance problems. The trend appears to be toward increasing use of tracing in production environments, with strategically placed static trace points being permanently enabled. Recent research describes the current focus on tracing technologies as being the result of increasing complexity in hardware and software (Desnoyer, 2009), and others have described the insight into the complexity of the operating environment as the key driving force behind their distributed tracing design and successful adoption of the solution (Sigelman et al., 2010).

While there are some similarities between these different sources of performance data, there is a mismatch in semantics and data volumes making this an interesting problem for research. Combining the two approaches using a unified framework for performance analysis is expected to provide additional insight that cannot readily be obtained if analysis is restricted to only one of the available sets of performance data.

This research has extended an existing open source framework in such a way that both arbitrary sampled values and arbitrary event traces can be observed and analysed using a common set of generic tools. Case studies evaluating the effectiveness of this approach are presented. Problems encountered along the way are also described, along with findings about the (sometimes unexpected) requirements of this unified approach.

Review

Classic approaches to a range of analytical approaches for performance modelling, bottleneck analysis and performance diagnostics (Jain, 1991 and Gunther, 2004) rely heavily on statistical sampling techniques. The many ways to approach problem solving using such methods is a mature research field, and thus we will focus our review on the emerging field of event tracing and the systems that have explored it.

Particular attention is paid to the visualisation techniques that others have explored to date, as a unified visualisation tool is expected to be important to our approach.

Magpie

Advances have been made in recent times in the use of event tracing to perform fine-grained system-level performance analysis. One of the earliest and most comprehensive event tracing frameworks is Magpie (Barham et al, 2003 and 2004). This project builds on the Event Tracing for Windows infrastructure (Park et al, 2004 and 2006) which underlies all event tracing on the Microsoft Windows platform. Magpie is aimed primarily at workload modelling and focuses on tracking the paths taken by application level requests right through a system. This is implemented through an instrumentation framework with accurate and coordinated timestamp generation between user and kernel space, and with the ability to associate resource utilisation information with individual events.

The Magpie literature demonstrates not only the ability to construct high-level models of a distributed system resource utilisation driven via Magpie event tracking, but also provides case studies of low-level performance analysis, such as diagnosing anomalies in individual device driver performance. Magpie utilises a novel concept in behavioural clustering, where requests with similar behaviour (in terms of temporal alignment and resource consumption) are grouped. This clustering underlies the workload modelling capability, with each cluster containing a group of requests, a measure of “cluster diameter”, and one selected “representative request” or “centroid”. The calculation of cluster diameter indicates deep event knowledge and inspection capabilities, and although not expanded on it implies detailed knowledge of individual types of events and their parameters. This indicates a need for significant user intervention to extend the system beyond standard operating system level events.

One major issue which we hope to tackle is identified in the Magpie literature during discussion of the use of Magpie in database performance analysis. Using event tracing, Magpie is able to track processor utilisation and I/O submission patterns. However, database servers make use of a shared buffer cache resource, and this is often a critical factor affecting overall system performance. Such caches are of course not unique to databases and caching is used in many high level applications to avoid interactions with slow components, whether that be tertiary storage or a second or third level cache. It is presented as an area for future work for Magpie to augment the existing traces with more information about the state of SQL server caching

during tracing. This approach would seem to indicate a more general requirement for increases in the volume of trace data as higher level caches are added, and again the requirement for detailed knowledge of higher level constructs at the lowest level (individual event traces). Cache state and cache effectiveness information is often maintained in higher level (typically sampled) performance metrics, and the approach of augmenting trace-based analysis with this system level information, rather than incorporating it into each and every trace seems a worthwhile trade-off. Certainly, while it may indeed be prudent to add some cache access information to an individual trace, one cannot possibly hope to know about and include the state of every possible cache affecting an event into individual event traces.

As an aside, it is worth noting here that, for the first time, we see in Magpie the use of a binary tree graph to represent the flow of control between events and sub-events across distinct client/server processes and/or hosts.

Although not explicitly stated (and relying heavily on the trace implementation), Magpie represents an early and successful foray into unifying sampling and tracing techniques in realistic production environments. The association of resource utilisation, a traditionally sampled class of performance metric, with events closely mirrors some of our goals. Our intentions deviate from it with the arguably more pragmatic and inclusive use of all available performance data in the system. This includes data from alternate operating systems and event tracing implementations, which are typical of distributed and heterogeneous systems.

DTrace

Arriving in the midst of the current wave of tracing technologies was the Solaris DTrace toolkit (Cantrill et al, 2004). DTrace provides dynamic tracing facilities, aiming to avoid the human overheads associated with static tracing. This is done in such a way as to not require any fixed kernel probe points initially, and to leave the system as it was originally (with no probe points or other modifications) once DTrace is no longer active. While it has since been shown that for tracing of some production workloads a small number of “always on” static trace points is warranted, DTrace represented a quantum leap in terms of both exposing a huge number of previously invisible trace points, and in greater flexibility by allowing almost arbitrary actions.

This is achieved through a high-level control language (the “D” language) which coordinates activation of probe points with trace actions. Built into the system is a focus on providing absolute safety in these user defined actions, as well as reliability in the reporting mechanism (dropped events are reported). The language significantly expanded a user's ability to control event reporting and propagation at the source of the event through use of predicates. This event filtering concept is present to some extent in most modern trace systems, but DTrace focussed heavily on the use of user-defined and thread-local variables, associate arrays and aggregations to augment these filtering capabilities. Furthermore, the flexibility of the tracing language meant that optimisation for bulk trace data transfers from the kernel into userspace was less critical. This reflects the origins of DTrace as a kernel tracing facility, with adaptations to also trace userspace activity.

Actions are prevented from modifying the system under observation, in all but a few situations and in well-defined and regulated ways. The system is dynamically instrumented such that when an event occurs, control flow transfers to an entry point in the DTrace framework which executes the handler with interrupts disabled, and finally control returns to the original call site. Constructs like loops are not representable in the D language (which allows only forward branches), reflecting the focus on safety.

Multiple consumers of events are handled transparently by the framework, not by the event provider. The consumer is expected to minimise dropped events by reading accumulated events in a timely fashion. An option to increase the size of the per-processor event buffers is also available. These higher level concepts (multiple consumers of individual events, reliable handling of data streams, making event drops visible to consumers, and having flexible buffering mechanisms in terms of memory allocation) are recognised as important requirements and must be reflected in a unified model for tracing and sampling.

Of particular interest in terms of our unified system-level tracing and sampling goal, is the DTrace concept of an *unanchored* probe. Unlike the other probes which generate events as the flow of program control passes (executes) them, these probes are associated with an asynchronous source. This includes a timer interrupt, which can be used to sample almost any data in the system and infer system-wide behaviour. This allows data sampled in this way to be associated with this self-generated event and made available to consumers.

The DTrace literature presents a detailed case study of the use of DTrace to diagnose a problem originally identified through the use of a sampling based tool – *mpstat*. This matches the general perception presented

here and observed in practice, that tracing is often a second-level approach for fine-grained detailed analysis, but often the trigger for analysis is a higher level anomaly detected using sampling-based techniques. In this case the DTrace developers actually began their more detailed investigation by tracing every single invocation of the code which incremented the sampled counter – a testament to the extreme versatility offered by DTrace.

DTrace had its beginnings as a kernel level tracer, and this remains evident today. While other tracing infrastructures focus on getting data out of the kernel into userspace, DTrace probe events are processed in the kernel for both user and kernel level tracing. Other tracing systems (Desnoyers, 2009), are able to achieve significantly more efficient userspace focussed tracing, with less disruption to the traced process, by processing the events within the context of the traced process (Scott, 2011).

A primary focus of DTrace is live system problem diagnosis and performance analysis. It is common in the traditional sampling-based performance tools and in many of the other tracing infrastructures to provide a retrospective analysis capability as well, often leveraging the same tools or presentation layer to assist user familiarity - no such effort has been reported for DTrace. This could be seen as a reflection on the original primary kernel level instrumentation focus. Alternatively, it indicates that a design focusing on processing events in userspace is more amenable to permanently recording data and providing both forms of information seamlessly.

One final noteworthy advancement made by DTrace is the concept of speculative tracing. This allows a trace to be speculatively retained for a short time, after which it must be committed to the trace buffer (and made available to consumers) or discarded. This feature is aimed at further filtering the event data at the source. One situation where this may be useful is to generate a single event where information from the start and end of the event is available together, such as the parameters to a system call and its return code (allowing filtered events of failing calls, for example, with the original arguments to the call included in the event).

SystemTap

Sharing many of the features and design goals of DTrace is the Linux SystemTap tracing toolkit (Prasad et al, 2005). Some of the stated aims of this system are to lift perceived DTrace restrictions, while attempting to maintain critical features like stability and safety in production environments. Other differences are a result of the nature of the targeted platform – design decisions taken for tracing the Linux kernel were seen as different to those required for the Solaris kernel where DTrace was born.

SystemTap introduces additional language features – including procedure declarations and a general purpose looping construct, which are not present in the DTrace language. This reflects one of the SystemTap design goals which is to aid kernel debugging, and to this end it incorporated many more aspects that allow modification of the observed system (writing to memory, invoking kernel functions) when run in a trusted “guru” mode.

X-Trace

Some approaches focus on tracing one specific aspect of a distributed system. Such is the case with X-Trace (Fonseca et al, 2007, 2008, 2010), which is described as a pervasive network tracing framework. While the focus is clearly on tracing network requests, it is done in such a way as to encompass tracking requests through the entire networking stack, from beginning to end. Due to the primary focus on networking and end-to-end request tracking, X-Trace has enjoyed some popularity in the web space.

X-Trace shifts away from the transparent approach typified by the dynamic tracers, and instead moves back toward extensive modifications of the instrumented components. Such is the nature of the network tracing problem – in order for X-Trace metadata to be maintained across arbitrarily networked systems, modifications (albeit optional) and extensions have been made to fundamental networking protocols such as HTTP, TCP, IP, DNS, and SQL to name a few. Several novel approaches have been used to augment these protocols so as to minimise the level of disruption in adding instrumentation, such as the passing of metadata in specially formatted SQL comments, passed between communicating systems.

The trace information can be examined, logged and reported at different points by different people in different administrative domains. Thus, the person (or application) requesting tracing is totally detached from the person receiving trace reports. The concept of a unique trace identifier is introduced, which is generated at the first instrumentation point and injected into every instrumented protocol packet.

It is a key design principle of the X-Trace system that it can use the trace identifier to reconstruct a “task tree” of sub-operations making up a task. Thus, X-Trace adds metadata into the same data path that it is tracing. However, the collected trace data is recorded and presented using mechanisms that are detached from the original data path. This protects against failure to report in the relatively common situation of network component failure.

Two sample visualisations for network event traces are presented within the X-Trace literature.

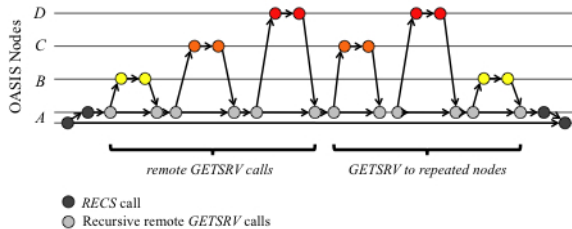


Figure 1: X-Trace Node Visualisation

Figure 1 shows an individual task trace tree, with several nodes involved (systems A through D). The X-trace visualisation shown in **Figure 2** is more typical though - presenting a top-down and left-to-right tree, with traces at the same network stack level shown horizontally and any sub-tasks (identified by edge type of “down” and parent identifier) immediately below.

These then may have tasks at the same level with edge type of “next” displayed horizontally with the preceding horizontal node identified through the “parent” identifier.

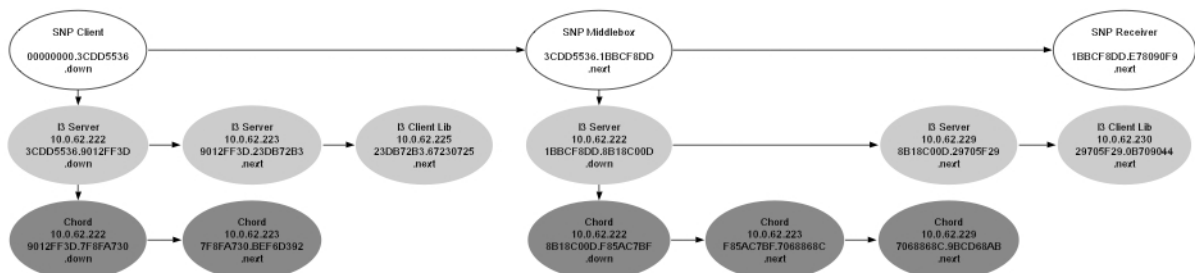


Figure 2: X-Trace Flow Visualisation

Another contribution to generalised tracing that X-Trace makes is evident in its metadata that aims to place the minimal necessary mechanism within the network to allow reconstruction of the path travelled. The metadata includes a “task identifier” (a task is analogous to the concept of an individual request from earlier tracing toolkits) – this field is not optional, and is replicated between networking layers to aid in preventing layering violations along the path. An optional three-tuple can also be propagated which is used to convey aspects of the trace tree. The three fields are a parent identifier, an operation identifier and an “edge type”. The two identifiers encode edges in the trace tree, and are unique within a task (request). The edge type indicates whether the metadata connects two adjacent nodes at the same layer, or between a node at one layer with a lower layer tree node.

As shown in **Figure 3**, the X-Trace metadata consists of:

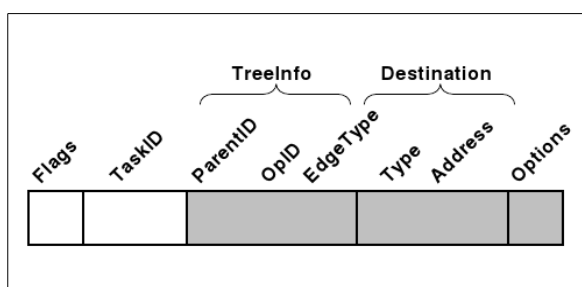


Figure 3: Structure of X-Trace metadata

- An initial flags field, which is primarily used to determine the length of the metadata structure based on the task ID size and which optional fields are active
- A task identifier (must be unique, and as small as possible)
- An optional field, *tree info* which records causal relationships in an operation identifier. There are three components to a tree info structure –

the parent identifier (linking traces together), operation identifier (component identifier within an operation) and edge type (horizontal/vertical)

- Finally, an optional *destination* field exists for sending trace data to specific loggers or other interested parties (if the reporting administrative domain chooses to allow that)

This metadata is the basis of the entire framework and must be as small as possible to attempt to address the requirement of minimal intrusion. In X-Trace the metadata is replicated across layers, usually in the “optional” fields of protocols designed for extension, which is the majority of the common protocols.

Of significance from a unified tracing and sampling point of view – the X-Trace literature identifies the need for trace reporting systems to add global system state into the trace reports. Although somewhat vague in this regard, it describes the need to associate particular examples of sampled data (such as “server load” with HTTP traces, and “queue lengths” with IP packet traces) alongside the trace data it is presenting. No attempt to encode that information alongside or within the trace data is made, however, and how to address the problem of associating the two forms of performance data is left up to the presenting device at the various layers of network stack tracing.

As with all tracing systems, X-Trace provides mechanisms to limit the amount of data traced and reported. Several techniques are described – sampling, batching and compression. Sampling is done at the point that a complete task tree has been formed, not before, in order that the tree structure is not lost. Batching refers to the batching up of many complete trees before forwarding on to reporting nodes. Compression is performed on the batched trace trees, prior to sending on for reporting.

Also, as with the other trace systems, handling loss of traces receives special treatment. This occurrence is particularly problematic in X-Trace, as it results in loss of nodes and edges in the reports, severely limiting their utility. Unlike the other trace systems, however, precision timestamps are less of a focus for X-Trace (notably, timestamps are not included in the metadata), as it targets more the identification of causal relationships and paths taken between networking components. However, this is identified as a limitation in the presence of lost trace data – if timestamps were present, an attempt could still be made to present the partial trace data on a common time axis.

LTT, LTTng, and UST

The Linux Trace Toolkit – LTT, or LTTng (“next generation”) in its current form – is a kernel level tracer specific to the Linux kernel (Desnoyers et al, 2007, 2009). Its focus is low overhead static tracing, and high-throughput extraction of event trace data from the kernel into userspace. It has since been augmented with a separate userspace trace (UST) facility which operates with low overheads in userspace, including lockless trace buffer updates.

A compelling case is made for very low overhead static tracing for the Linux kernel (Bligh et al, 2007) through a series of real-world case studies. Although it appears to conflict with the direction taken by the dynamic tracing approaches favoured by others, having a wide net cast over an existing set of critical core kernel functionality is demonstrated to be useful for getting an initial feel for the area in which a problem lies on the first time that it occurs. Subsequent iterations on attempting to find the problem are facilitated through quick, simple instrumentation extensions adding to the existing set. Static tracing provides better access to local variables and hardware registers, and the use of lockless algorithms have allowed the tracer to no longer have to disable interrupts (lowering overheads and opening up additional potential trace points). This approach does require the kernel to be rebuilt, redeployed, and machines restarted. However, the focus on fast timestamps and low impact tracing allows debugging of classes of problems, such as subtle race conditions, that while very hard to hit and debug become more of a recurring problem when the number of machines involved increases. This is the pathway of “always on” event tracing, which can complement the previously reported dynamic tracing mechanisms.

In addition to setting the benchmark in exploring the extents to which tracing overheads can be reduced, the LTTng literature describes other interesting tracing techniques. In kernel tracing, the ability to have tracing enabled at all times but to stop when a particular event or condition is detected, to prevent overwriting of the trace buffer data with new traces is described. This is used in a so-called “flight recorder” or “overwrite” mode, where trace data buffers of fixed size are continually overwritten by newer data until the critical condition is detected. Further practical tracing lessons from the Linux kernel environment are presented, with description of how traces help to communicate problems amongst developers from different organisations and backgrounds to be able to “see” the details of these very difficult to reproduce scenarios.

An approach to recovering (sampled) system metrics from kernel traces has been explored (Giraldeau et al, 2011) using LTTng. This paper provides insight into the divide between kernel developers and system administrators which is of interest here. Event traces tend to expose low-level kernel internal information, requiring a depth of domain-specific knowledge that many system administrators do not have. Examples of being able to reconstruct the state of certain information off-line from trace records on disk are presented. The paper perhaps inadvertently makes an excellent case for the need for unified event trace and sampled performance data, as this entirely removes the need for such expensive off-line post-processing.

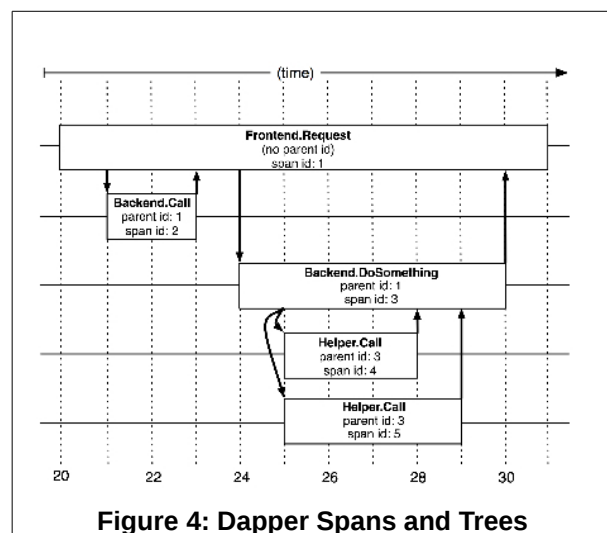
The LTTng focus remains on single-node tracing, and it is reported as successful in that space. However, the need for tracking dependent events between clustered nodes and to follow dependencies between the nodes, is explicitly noted as future work in several papers. Amongst the requirements listed for such extension are low-overhead methods for aggregating data over the network for analysis, sufficient information for analysing inter-node communication and an analysis tool capable of presenting inter-node relationships and displaying multiple parallel traces.

Dapper

An evaluation of several years of experience from developing and using a large scale distributed production tracing system is presented in Dapper (Sigelman et al, 2010). This system uses an enforced, minimalist set of correlated and distributed request traces, coupled with a simple API for higher level developers to annotate traces. It operates in a largely homogeneous production environment. Dapper makes use of a global identifier to associate related trace events within a distributed environment, similar to the Magpie and X-Trace systems described earlier.

One of the cornerstones of the Dapper design is the “always on” tracing approach. Everything that can be traced by Dapper is always traced, there is no concept of conditional activation. This requirement clearly reinforces the need for minimal-cost tracing infrastructure. The implementation addresses this through software efficiency, and through the use of a relatively small number of trace points at critical flow control points. Furthermore, reduction of events via adaptive sampling at a higher level is performed for very high traffic situations, some time after the event has been consumed but before it is further propagated or committed to persistent storage.

As shown in **Figure 4** Dapper traces consist of trees, spans and annotations. The nodes of trace trees are named “spans”, and each is a set of timestamped records encoding start and end time, possibly remote procedure call (RPC) timing information, and optional application specified annotations. Edges of the tree indicate a causal relationship between a span and its parent span. In addition to a human readable span name, spans also have an individual identifier and (optionally) a parent identifier. All spans associated with a trace also share the trace identifier. Note that it is typical for a span to contain information from multiple hosts – an RPC span contains information from both client and server processes. Timestamp skew between machines is noted as an issue, as it was for proposed LTTng extensions in distributed tracing.



The Dapper system, like X-Trace before it, performs trace logging out-of-band with the request tree itself. This avoids affecting network dynamics, and avoids the assumption that RPCs are correctly nested (not the case if a result can be returned to a caller before all backends have completed).

The issue of security and information privacy is highlighted, and some of the issues are similar to those described for X-Trace – all trace systems must consider this issue, as trace event parameters will often contain sensitive information. The Dapper approach is to not record any payload data unless requested via application level annotations.

The Dapper literature describes common user access patterns to the long-term stored trace data. Access by trace identifier is common. Bulk access to all traces in parallel is facilitated by a map-reduce interface. Indexed access is provided to map from commonly requested trace features to trace identifiers, and the choices of index are noted as having been challenging, due to the increase in required storage. Two indices were initially implemented – trace identifier lookups using host names and service names – however, the former was later removed due to insufficient user demand (relative to the storage costs). Concluding from one particular example, a relevant issue in this context is highlighted – it is evidently non-trivial to integrate Dapper traces with general-purpose log file traces.

Other limitations identified in the paper include coalescing effects (such as buffered disk writes), where an individual traced request can be “blamed” inappropriately for a large unit of work, or where traced requests are batched and processed together. The ability to perform root cause analysis is questioned – where higher level distributed system behaviour is relatively easily observed, effects such as delays observed in individual traces due to outside queuing effects are not. One possible solution that has been used is to add application annotations to traces describing these parameters, but the problem is systemic and unlikely to be either satisfactorily or efficiently resolved with this technique. Finally, associating trace information from other tracing environments, such as the kernel, is highlighted as an area of need in Dapper. Difficulty has evidently been observed in correlating the userspace trace instrumentation with the trace instrumentation from the kernels of individual nodes in the distributed system under analysis. A solution involving snapshots of a few kernel level metrics to associate with an active span was under consideration at the time of publication.

Intrusion Detection

In the security research domain, some developments relevant to our project are worth review. These developments involve detection of anomalies in logged or traced data, with the aim of identifying security breaches. Of particular interest is a system that combines visual and automated data mining for live anomaly detection (Teoh et al, 2004) for use with the Border Gateway routing protocol (BGP). This system gathers routing data much like we gather traced performance data, then filters and processes it to obtain statistical measures of anomaly for each routing update message.

Routing data is selectively transferred to the visualisation client based on a routing domain of interest, and can select both live and historical data to be visualised together. Along with each cluster of update messages, the anomaly measures are also presented using a novel “event shrub” visualisation which resembles a horizon dotted with trees (“shrubs”) as show in **Figure 5**. The stalk of each “shrub” pinpoints time locations, and the circular “foliage” has size and colour variations indicating the nature of the statistical or signature-based anomaly measures calculated for individual route update events.

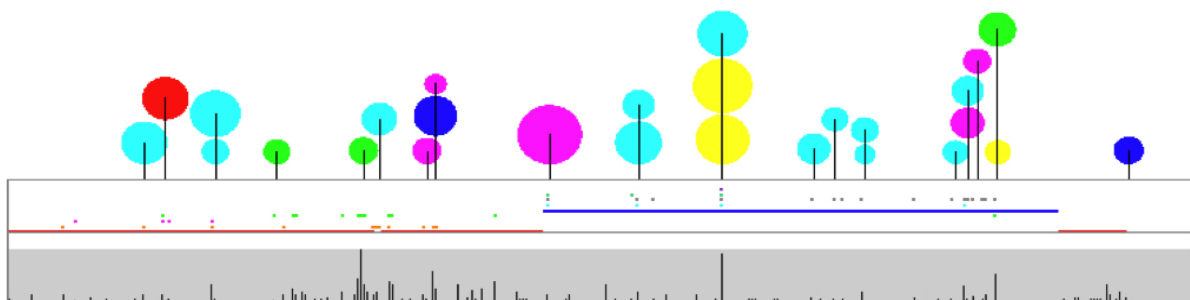


Figure 5: Intrusion Detection Visualisation

NetLogger

Close to the level of analysis we are targeting is the NetLogger system (Tierney et al, 2003). Originating in the high performance computing grid sector, this toolkit uses timestamped event logs to track end-to-end monitoring of applications and the underlying operating system. The approach aims to identify high latency outlier requests and unexpected low throughput, and is based on a methodology of structured application-level logging. Generated logs consist of structured, self-describing text records and although the inefficiencies of this format are discussed (many fields are repeated, and the use of text format produces

larger data volumes), the authors initially propose that the simplicity of processing outweighs these downsides. Noticeably, this is in contrast to many other event tracing advocates we have reviewed, and was likely influenced by design simplicity when working with arbitrary logged (text) data. Furthermore, in more recent versions of NetLogger, a binary protocol is introduced with apparent orders of magnitude performance improvements.

NetLogger advocates the use of a hierarchical namespace of unique names for each logged event. The use of a “levels of detail” concept as an option for filtering at both the event source as well as at the event consumer is recommended. Finally, their concept of (trace) identifier is open-ended and flexible – supporting both globally unique identifiers as well as the native identifiers used with application software (such as request and process identifiers). The trace toolkit should not be rigid in how it allows traces and sub-traces to be identified, as it is highly advantageous to identify and report on entities using the local application terminology.

In addition to the event log (tracing) components of NetLogger, wrappers for standard Unix system and networking tools (vmstat, netstat, iostat, snmpget) are used to augment the log streams. This is given relatively little attention in the literature, however, and the primary focus appears to be around interpreting the event log traces. Several notes are worth making with the wrapper based approach – it is simple and requires relatively little maintenance. However, it assumes a fixed sampling rate at the monitored host (each wrapped command will have its own mechanism for specifying sampling frequency). Some difficulty may be observed with data coherency, since each wrapped tool will be operating independently. While similar frequencies may be achieved, it is highly likely that the samples are taken with (possibly significantly) mismatched timestamps within the sampling interval.

The NetLogger visualisation tool uses three types of graph primitives to represent different events. It names these types “points”, “loadlines” and “lifelines” (each type represented in **Figure 6** below). Time is displayed on the X-axis and ordered events are shown on the Y-axis.

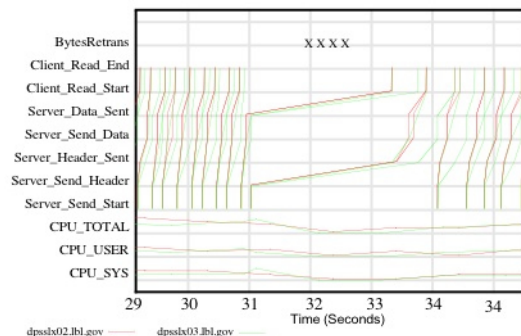


Figure 6: NetLogger Lifeline Visualisation

The gradients of the lifelines show latencies of events being traced through the system, and they are generated by correlating trace events by timestamp and event identifiers.

A loadline is effectively a line plot, and connects values in a continuous segmented curve. It is typically used to plot sampled data like processor utilisation (final three plots in **Figure 6**). The point plot is used to display single occurrences of events. In this example (first plot in **Figure 6**) it shows retransmitted network bytes, which is a sampled kernel performance metric, but it could clearly be used for event trace data as well, where the events are singular one-off events (e.g. SNMP traps), without distinct start and end times, and without an associated trace tree.

Mixing Metaphors

Prior research into the area of combining statistical sampling and event tracing is relatively limited – almost all work focuses on one approach exclusively. In the few cases in software systems in wide-spread use where both approaches are supported (such as the Intel *Vtune* profiler, and the Windows *logman* utilities), this is primarily done such that one technique is activated at a time to the exclusion of the other.

Combining system level sampling with distributed event traces is identified as an area of need for future research in the Dapper system (Sigelman et al, 2010), which presents a case study of a very large scale distributed tracing infrastructure.

An investigation into combining the techniques has been performed in the embedded system space (Metz, 2004). While more focussed on enhancing the profiling of activity in low powered embedded devices, this work is highly relevant to distributed system tracing as well. The hybrid approach described there suggests balancing event tracing and statistical sampling through limiting the profiling data volume. This is achieved by restricting profiling to short, focussed runs involving capture of trace data, and recording the remainder of performance data through statistical sampling. This would appear to conflict with the “always on” requirement for tracing advocated by others more recently (Sigelman et al, 2010). The work precedes the introduction of the concept of dynamic tracing, which was first published in the same year, listing the static nature of event trace instrumentation as a major issue with the approach.

The details of the implementation of their approach are not specifically presented, unfortunately, but they do state that it produces “two separate sets of profiling data [and] these two sources of information need to be combined and synchronized during post-mortem analysis”. This leads us to believe there are significant differences in their approach to ours, which combines the performance data at sampling and event generation time, and maintains a unified data stream. Despite these differences and a lack of detail as to their approach, this paper provides us with good clarification of the issues and the knowledge that a combined solution is indeed sought after even in these smaller scale embedded systems.

Performance Co-Pilot (PCP)

The PCP toolkit (McDonell, 1999) provides a system-level performance analysis toolkit suited to distributed system analysis. It transparently supports live and retrospective analysis, and it aims to allow data from each component of a complex system to be incorporated into the analysis framework.

The mechanism through which individual components of a complex system are incorporated is agent-based. Each domain of performance data (hardware, kernel, database, application, services, and so on) typically has a single performance metric domain agent (PMDA) associated with it.

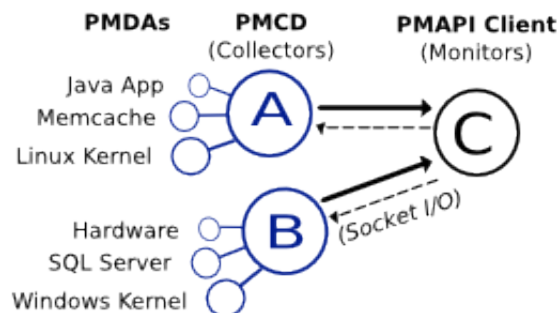


Figure 7: Distributed PCP Components

These agents export the current value(s) of performance metrics on request. A performance metric may be a singleton (such as the number of context switches) or set-valued (such as the number of read operations to each physical disk).

Individual performance metrics are strongly typed. Each must be one of:

- a **free-running counter**
Monotonically increasing value, such as the number of read operations to a database, or number of bytes sent across a network interface.
- an **instantaneous value**
Point observations, such as the amount of memory in the kernels page cache, or the length of a device driver queue.
- a **discrete value**
Relatively constant value, such as the amount of physical memory or perhaps the maximum number of threads configured to service requests in an application.

- an **event record** stream

A compound metric, which allows event traces to be encapsulated, such as records arriving in a log file, or events generated by tracing libraries.

The basic counter, instantaneous, and discrete types of performance metric, above, have proven sufficient to capture the essence of *sampled* performance data from complex computing environments. The values of these metrics are instantiated at an interval driven by the client (monitor) analysis tools. It is extremely rare that any state need be maintained for sampled metrics on the instrumented system. Rate conversion of counter metrics does require the value from the previous sample, but importantly this state is kept by the monitor tools.

The event metric type is a relatively recent addition. Individual event records have a timestamp and parameters that are associated with the event when it occurs. Event timestamps are not driven by the client analysis tools sampling rate at all. However, within a given sampling interval, any number of event records may be generated, and the event type metric provides the mechanism by which these events (along with their timestamps and parameters) are propagated to interested analysis tools, possibly simultaneously with sampled metrics.

Two special parameters can be associated with event records. An agent instantiating event metrics can choose to decorate the events with certain flags. The initially defined set of flags allow structural relationships between distinct events to be represented. In particular, a parent-child relationship can be represented, as can begin-end event pairs. The parent-child relationship allows arbitrary identifiers to be represented. Identifiers name the event source – for example, a process identifier (numeric) might be used, or perhaps a web request identifier (character string). The second special parameter is an optional count of missed events. This provides a mechanism by which a client analysis tool can be informed that they are not keeping up with the rate at which events are being generated.

It is the primary purpose of this research to investigate and instrument real world problems with both sampled and event trace-based metrics, and evaluate the effectiveness and difficulties in unifying these metrics.

Unified Metrics Namespace

PCP provides a hierarchical namespace of performance metric names which can be queried by analysis tools (clients) to discover the complete set of available metrics from a collector system. These will very often differ between even homogeneous systems, and certainly between different operating systems. The model for a typical monitor / collector exchange is as follows:

- monitor tool explores the available metric namespace hierarchy
- collector service responds with parts of the namespace (leaf and non-leaf nodes)
- monitor requests numeric performance metric identifiers for those metrics it chooses to observe
- collector responds with metric identifiers for leaf nodes (metrics)
- monitor requests additional metadata about chosen metrics
- collector responds with metric descriptors, which provide metric types, semantics, units, and so on
- monitor requests values for chosen metric identifiers
- collector responds with current values (sample taken or all event records from the previous interval)

For long-running client tools, the final two steps are typically performed in a loop and analysis or reporting is subsequently done with the latest returned values on each iteration. In the case of retrospective analysis, the same basic model is used except the collector is a time-indexed archive of performance data instead of a live system.

Also, the above model is a streamlined version for the sake of simplicity, which ignores the slight complication presented by set-valued metrics. For that case, individual values can be named, and there is an additional exchange involved with discovering those names and for restricting the set of returned values to a subset.

Note that the monitor tools requests for values are simply using sets of metric identifiers – this proves to be a lightweight yet still flexible and dynamic requesting mechanism. It allows an individual client to request different metric values on different intervals, and the resultant values always dwarf the request. With event metrics in the request mix, this ratio is amplified (potentially significantly).

Sampling-based Agents

The role of a performance metrics domain agent (PMDA) is to respond to requests from a performance metrics collection daemon (PMCD) process, on behalf of client processes. Each is focussed on one *domain* of performance data – the kernel, a relational database, an application, a particular class of hardware, a web server, for example. There is usually a one-to-one mapping with the requests outlined in the basic request model earlier for the collector system. So, each PMDA has complete control over the set of metric names, values, semantics, types, units, and so on that it exports. Different agents respond to requests in different parts of the metric namespace, and this division of labour is arbitrated by the PMCD daemon.

PCP includes a large number of agents out of the box; packaged, tested and ready to make metrics from their domain available to the analyst. Extension to individual needs is actively encouraged (and highly advantageous) using the PMDA shared library and API.

As was mentioned previously, the mechanics behind sampling-based metrics rarely require state to be maintained on the part of the PMDA. Agents using sampling only are thus far simpler than those using event-based metrics. It is important to note that both classes of metrics can co-exist within a single PMDA, through the use of distinct metrics.

Sampling-based Visualisation

Representing sampled performance data visually is a well trod path, and the utility of line plots, stacked bar plots, scatter plots and histograms are well documented. Much of the reviewed trace literature devotes time to presenting different aspects of visualising trace data, in particular in representing trace start and end times but also in presenting causal relationships between components of a complete trace.

The PCP *pmchart* tool depicted in **Figure 8** demonstrates some common and powerful user interface techniques that have been traditionally used in the sampling space. These include stacked bars and line plots, with the X-axis showing changes in time, and the Y-axis displaying value ranges for the plotted metric values.

Note that this strip chart utility allows visual correlation of performance data from different domains – here we show processor utilisation (physical hardware domain) visually correlated with virtual machine counters (virtualisation driver domain).

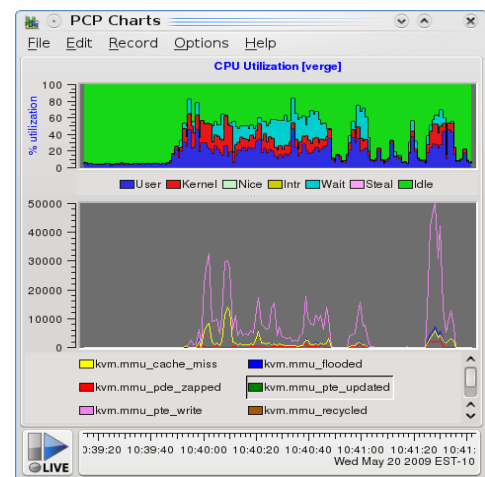


Figure 8: Interactive Strip Charts

Method

There is no unifying theory to guide this investigation, so a pragmatic approach has been taken. Like most prior work in this area, the approach for assessing effectiveness of a potential approach is to build a system which instantiates the central ideas and provides the desired features. Evaluation then proceeds in a realistic production environment where a number of case studies are undertaken to identify the strengths and weaknesses of the approach. This environment is particularly important as the analysis system must endeavour to minimise the changes it makes to the behaviour of the instrumented system.

This research has been undertaken in much the same way. Due to the scope of unifying two approaches which have historically generated many individual projects independently, an existant system was used as a starting point for extension and further research. This is the Performance Co-Pilot (PCP) toolkit. It has been selected for several important reasons:

- long history of successful production deployments (over 15 years)
- open source project, so any new software developed can be evaluated in a peer-reviewed open environment, well-suited to research
- domain agent-based approach that lends itself to extension
- provides a clean separation of responsibilities between agents (collectors) and clients (monitors)

- stable and well-documented interfaces suitable for extension with new domain agents as well as new monitoring tools (and with clear separation of responsibilities)
- researcher access to production environments where it is deployed
- researcher familiarity with the software

Software Artefacts

A brief summary of the software produced during this study is warranted. A detailed rationale and analysis of each component is provided in the case studies following. The needs and opportunities presented by each case study dictated the direction that development work took.

On the collector side, several new PCP agents have been created (*pmdarsyslog*, *pmdabash*) and extensive modifications to an existing prototype (*pmdallogger*) has been undertaken. An API for PMDAs implementing event trace metrics was created, and existing code refactored to make use of it.

On the monitoring side, extensions have been made to the *pmevent* client tool to perform event filtering, and extensive modifications have been made to *pmchart* to experiment with a unified visualisation model.

Event Trace-based Agents and Queueing

The initial introduction of event trace metrics infrastructure into PCP (early 2010) brought about the need for agents to keep track of which clients have expressed interest in which event metrics they export. A connection-oriented protocol is thus highly advantageous for this approach. Not only that, but the agent must track which events have been sent to which clients so far, as the client tools may be distributed remotely on the network and may be requesting the latest set of values (both sampled and event arrivals) on different intervals. These interactions are shown in **Figure 9**.

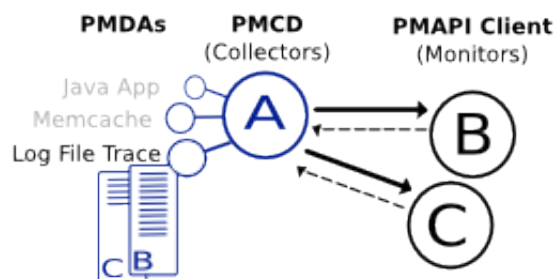


Figure 9: Event Trace Agent Queues

Further complicating matters is the need to minimise the extent to which the instrumented system is affected by being monitored. A direct requirement from this is a need to limit the amount of memory that may be used to hold buffered event records in the PMDA process before they are sent to clients. In the situation that events arrive more quickly than the clients are draining them, the “missed” event counter mechanism described earlier is used to inform the client of the number of events they missed out on.

Fixed sized buffers holding reference counted events (not multiple copies of the event data), and per-client tail-queue based data structures have been used with some success to implement these requirements:

- If no clients are expressing interest in an event metric when one of its events is delivered to the PMDA, then the event can be discarded;
- Otherwise, newly arriving events are added to the front of their respective queues;
- A pointer to the last (tail) event that each client analysing event trace metrics has seen (from the previous query) indicates where each client is up to in the queues;
- As each event is shipped off to a client, the tail pointer for that client is updated, and a check is made to see if that client held the last reference to the event. If so, after it has been sent the event can be safely discarded and the memory reclaimed from the fixed size pool.

Visualising Event Traces

Generic concepts common to several of the event tracing tools reviewed have been identified. These represent characteristics shared by all event trace metrics independent of performance metric domain. The Performance Co-Pilot *pmchart* utility has been extended to display event metrics using this new visual representation (**Figure 10**). The salient characteristics identified for further visual exploration are:

- Event identifiers and timestamps
 - A single point represents the presence of an event
 - All events sharing an identifier are mapped to the same Y-Axis value, and the same colour
- Parent-and-child relationships between events
 - A vertical line associates a parent event with a child event
- Start-and-end relationships between events
 - A horizontal line drawn from the start event to the end event using the same colour as all other events for that identifier.

Where these relationships between two events are being depicted, it is required that the events have identifiers which can be used to visually associate the different components. These identifiers are also used to distinguish events on the Y-axis of the graph. For this study a very simple algorithm mapping between event identifier and Y-axis value was used. Events which have no identifier are bucketed into a common Y-Axis point and rendered at the lowest point, all in the same colour. By definition, these have neither horizontal nor vertical connections, and are represented by points only.

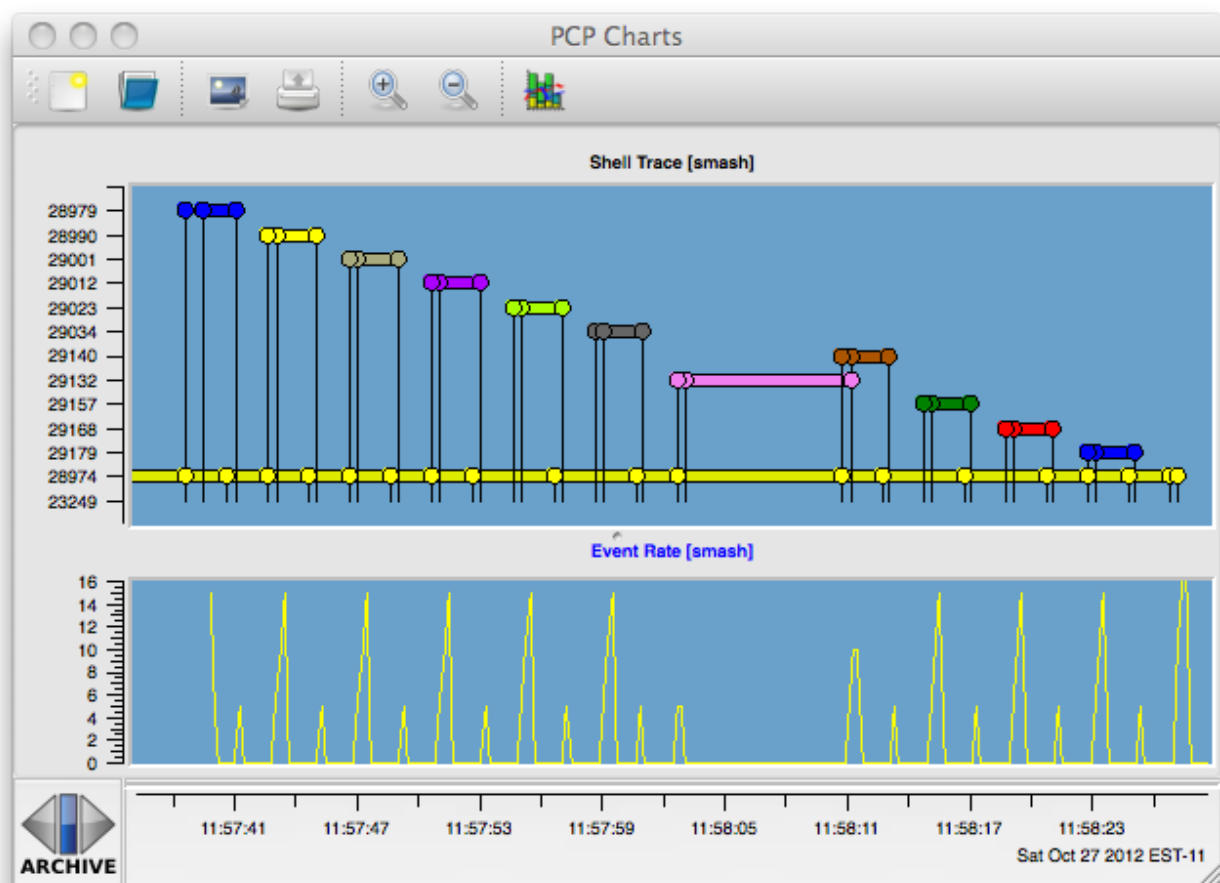


Figure 10: Unified Strip Chart Visualisation

As was reported in the X-Trace (Fonseca et al, 2007) and Dapper (Sigelman et al, 2010) environments, this visualisation fundamentally requires event identifiers in order to display a richer representation. The basic layout algorithm used was selected for its simplicity, far more compact representations could be achieved using more advanced techniques (force-directed layout, simulated annealing, etc.) but for the purposes of our study here, these were not seen as required and have not been attempted in this initial implementation.

With the foundation of a unified visualisation model now in place, we move onto the finer details of interactive performance analysis with the modified *pmchart* utility, as relates to our unification goals.

When viewing sampled performance data, the graph can be interacted with by increasing and decreasing the zoom factor. This allows a coarser or finer granularity of both event and sampled data display. Note that in the *pmchart* tool, as is typically in this space, this affects the X-axis (time) only.

Zooming in allows fewer values to be displayed in the same visual area on the display. In the sampled case, this tends to simply draw ones attention to a particular spot without allowing more information to be expressed visually. However, in the case of trace data it was found to have the additional, important effect of displaying more and more events. This is because the event traces typically are generated rapidly and with small time offsets, giving (at coarse granularity) the impression of just one or two events. As one zooms in, however, the small differences in the event timestamps become more pronounced and more noticeable.

Zooming out causes a wider range of X-axis values (time) to be displayed. For tracing, this has the effect of allowing the connections between related events be more easily discerned, while reducing the individuality of specific events (points) – that is, individuals are more likely to overlap with peers in close time-proximity.

The user is also able to gain finer detail about aspects of a sampled graph through use of the pointing device and selection of a point of interest. Because the values displayed in a sampled graph have both X-axis (time) and Y-Axis (metric value) significance, the user can be informed of a single metric value at the selected point. A cross-hair visual indicator is used, and the time/value pair is reported at that point. Pressing and moving the pointer provides a continual feedback of these time/value pairs. This has proved highly useful in practice.

The event trace graph, however, shows the trace identifier as Y-axis value. This is a discrete value, and does not exist in a continuous space as with sampling-based graphs. The data associated with an event (the event parameters) is more complex than a time/value pair. Furthermore, with the possibility of dense clustering of events it has been identified as desirable to be able to select not only individual events, but closely related groups of events. An extended selection model for event traces has been implemented for this reason, allowing groups of events to be selected at once using an “area picker” (depicted in **Figure 11**). Selection of one or more events results in visual feedback identifying selected events, as well as a secondary window being displayed with details of the selections (timestamps, and all event parameters).

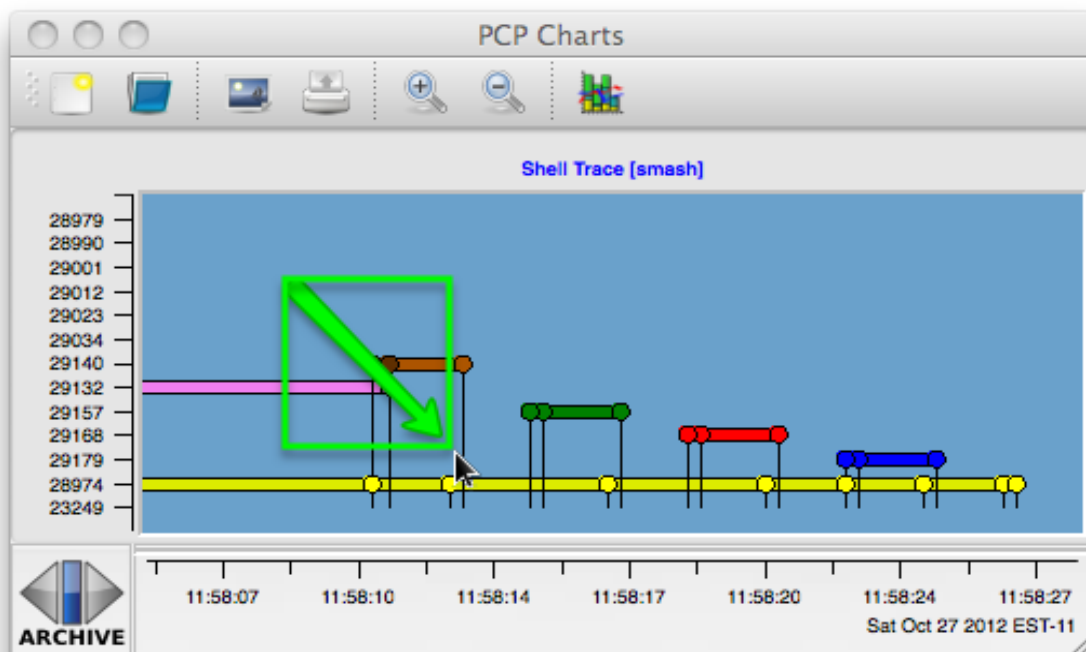


Figure 11: Extended Trace Selection Model

Log Streaming Case Study

Capturing system event log information from a collection of geographically distributed systems in a reliable and useful way presents many challenges. In order for operations and engineering triage teams to filter important events from this huge amount of information, events need to be shipped to a central location, reliably, where they can be indexed and searched. Searching is typically keyword and time range based.

A study of the log streaming present in a production web application has been undertaken. This deployment comprises a series of loosely connected machines, cooperating to provide a coherent service. Each machine has a system log, which coordinates kernel and system-level logs using the *rsyslog* software. Additionally, the web application itself has been configured to send its logs to the *rsyslog* daemon.

This system logging daemon has been instrumented to extract both event record and counter metrics.

Centralising Event Logs

Ideally log streaming is done in a way that is both timely and removed from the systems under observation. Being timely allows events relevant to an active production problem to be quickly available to triage personnel. Being removed from the production system reduces the impact on the observed system, and also allows for collation of events from cooperating systems (separate databases, application servers, web servers and storage servers, for example).

In addition to the events logged by the operating system kernel and the system daemons, it is also highly desirable to capture application events as well. For minimal operational maintenance overhead, these should all be managed by a single, reliable event shipping system for application logs.

This case study documents the design and deployment of one such system, and focuses on the performance instrumentation used for monitoring and problem diagnosis in the event management service itself.

rsyslog is the default system log daemon on most Linux distributions today, and it provides efficient, reliable end-to-end delivery. It turned out to be easily instrumented - providing both its own metrics and mechanisms for adding metrics specific to our needs. Both sampled and event trace metrics have been made available.

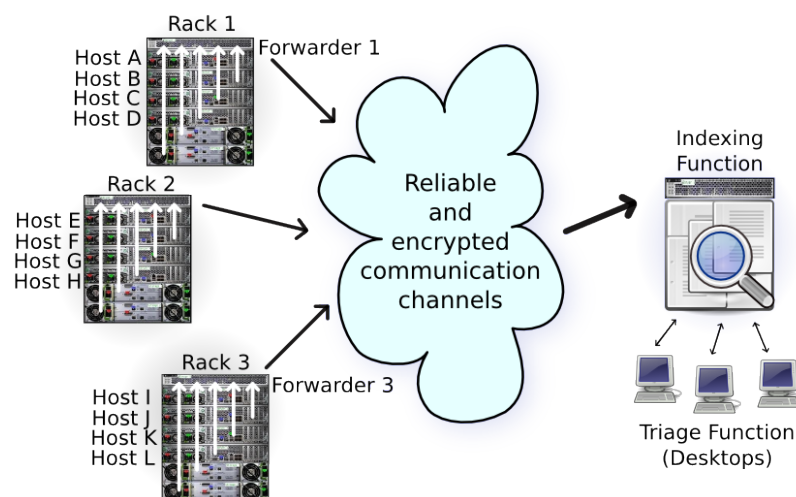


Figure 12: Log Stream Design

The design caters for a fairly typical medium-sized web application deployment. Each data centre hosting the application (globally distributed) contains several racks of equipment dedicated to delivering the service. As shown in **Figure 12**, individual racks are populated with closely cooperating machines, each generating their own system logs and application logs.

Deployment and Instrumentation

It was highly desirable to gain insight into many levels of the process around transferring logged messages. Identifying hosts generating too much traffic, or hosts not generating any log traffic (misconfiguration) was initially important - so, event counters, and cumulative byte counts of events generated was required. It was also important to be able to see these data rates alongside the network interface data rates, to understand the additional load generated through live streaming of the event log data.

The *rsyslog* daemon runs on each and every machine involved, so low overhead is a desirable attribute in any instrumentation added. Not all of the machines are configured the same way though, as can be seen in **Figure 12**. Every host in the system generates at least its own local system logs. The event "forwarder" systems (depicted at the top of each of the three racks) are sent all of the events generated by systems within a rack, and they then forward them on to the next node in the system (either another forwarder, or the indexing system). The event "indexers" (right hand side indexing host) are responsible for creating a searchable index of all event messages from all systems. So the configuration of *rsyslog* differed from host to host and a great deal of care was required in the roll out. Misconfiguration can result in forwarding loops, causing loss of functionality and overall network performance degradation.

On inspection, it turned out that there is existing instrumentation for the internal workings of *rsyslog*. It must be explicitly enabled - both at the source level and at runtime. To enable the instrumentation in an *rsyslog* build, the `--enable-impstats` configuration flag is needed.

A two-pronged approach to instrumenting the *rsyslog* processes has been taken. Firstly, the internal instrumentation has been exposed as PCP metrics. Secondly, in addition to the path the messages would usually take (to a local log file, forward on the network, etc.), all arriving message are also enqueued to local named pipes. Events (messages) arriving on these pipes are then turned into PCP event metrics.

To achieve these levels of instrumentation, the additions to the *rsyslog* configuration show in **Table 1** were required.

```
# Provide rsyslog statistics (pmdarsyslog)
$ModLoad impstats
$PStatsInterval 5
syslog.info                                | /var/log/pcp/rsyslog/stats

# Performance instrumentation (pmdallogger)
local0.*                                  | /var/log/pcp/logger/applog
*.*;local0.none;syslog.!=info            | /var/log/pcp/logger/syslog
```

Table 1: Configuring rsyslog

The first section instructs *rsyslog* to load the statistics module, and to export the current state of its statistics every 5 seconds. With the configuration shown, these are exported to the named pipe at `/var/log/pcp/rsyslog/stats`. This named pipe is drained by the PCP *pmdarsyslog* agent, which was created specifically for this experiment, and it exports the following PCP metrics:

<i>rsyslog.queues.size</i>	<i>rsyslog.imuxsock.submitted</i>
<i>rsyslog.queues.maxsize</i>	<i>rsyslog.imuxsock.discarded</i>
<i>rsyslog.queues.full</i>	<i>rsyslog.interval</i>
<i>rsyslog.queues.enqueued</i>	

["imuxsock": counts related to system log input messages on **Unix domain sockets**]

Figure 13 depicts the use of these sampled values to correlate logging behaviour with overall network traffic on an instrumented host. With existing knowledge of the load in the time ranges depicted, one can tell that between 14:04:20 and 14:04:44 an increase in the count of events enqueued to *rsyslog* is observed, and that these are not Unix domain socket events. Rather, they have originated from the outside network (arriving on network interface *eth1*). At this stage we have no deeper visibility into the contents of these messages.

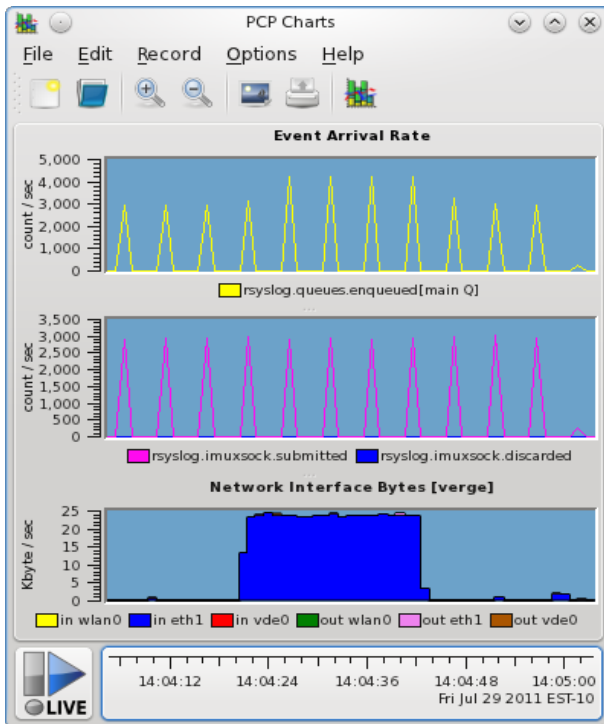


Figure 13: System Log Event Arrivals

observed events.

```
logger.perfile.applog.bytes
logger.perfile.applog.count
logger.perfile.syslog.bytes
logger.perfile.syslog.count
```

An interesting finding from beginning to use this event data is that simply counting the events as they arrive (in addition to making the event payload available), is trivial and highly useful. It turns out that a common problem with event records is understanding firstly whether they have arrived at all, and secondly being able to distinguish the difference between “no events are arriving” and “all events are being filtered”. This is made easy by exporting a global counter of events that have arrived at the PMDA.

Similarly, it is simple to add a second counter metric which exports the volume of event trace data that is arriving at the PMDA, and this is again extremely useful in practice.

Figure 14 shows these values being depicted in the PCP charting utility. The topmost graph (stacked bars) gives an indication of log volumes for the monitored *rsyslog* process (application logging in magenta, system logging in yellow). The lower graph shows the event rate (line plot) for application logs and system logs, as rate-converted counter metrics. The strong correlations here indicate that much of the system-level log traffic is being caused by application-level activity. This is relatively common in this case study – for example, a user interacting with the application may cause mail to be generated, which invokes the local *postfix* mailer, which generates system-level logging as it attempts delivery. This is a multi-step process, with multiple system log messages from a single higher level application operation.

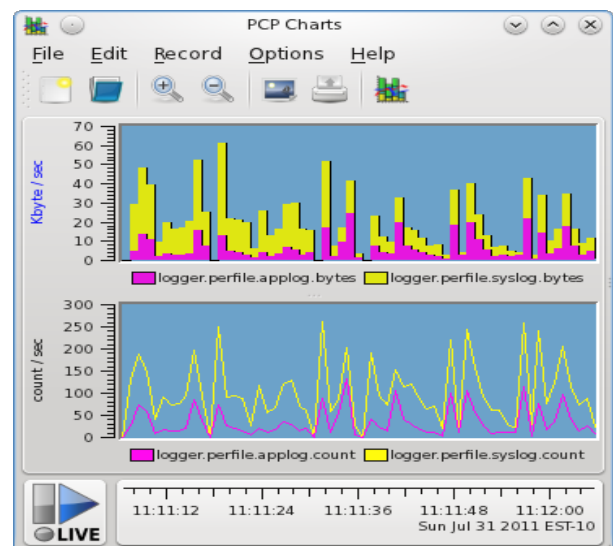


Figure 14: System Log Event Throughput

Event Analysis and Filtering

The *pmdallogger* PCP agent we mentioned earlier has visibility to all event traffic (log messages), and is separately counting application and system log traffic. It has been used to inspect log contents (for checking event arrival, for diagnosing host misconfiguration, and for verifying formatted message validity) in what proved a much more convenient way than snooping the raw network traffic. In addition to the (sampled) counter metrics described previously, and for the purposes of the current research it also exports the following event records as performance metrics:

logger.perfile.appllog.records *logger.perfile.syslog.records*

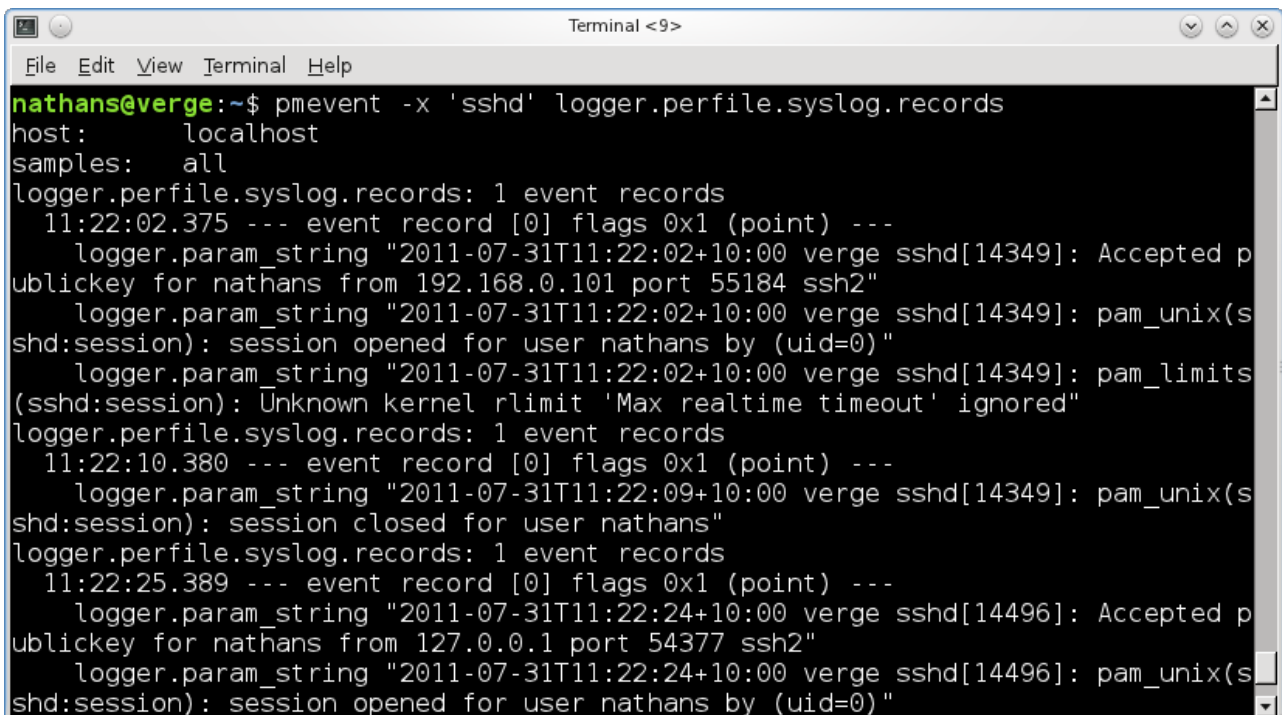
At this stage of the project, we only had a simple client tool for decoding event records (later in the project, a more powerful visualisation utility was implemented). This was the command line utility *pmevent*, which is the first and simplest event tracing tool. Even in this basic "text dump" form, insight was gained by being able to see the exact traffic passing through each output stream, and configuration problems were diagnosed and resolved through its use.

Here the importance of event filtering was emphasised in a practical setting. One issue making problem solving difficult using event data from individual events is the volume of data to sift through. When events are arriving rapidly, it is impossible to know which hold information of interest and which are noise.

In experimenting with ways to address this problem, the *pmevent* utility was extended with a new -x option (shown in **Figure 15**). It provides a mechanism for server-side event message filtering using regular expressions. A monitoring client (*pmevent*) sends a string to the collection agent (*pmdallogger*) for a specific metric. In this example, it allows the user to match only system log messages that are related to the *sshd* process, by applying a regular expression filter on the contents of the system log events.

The filter in the *pmdallogger* agent is maintained per-client, so other users also watching event metric values could specify a different filter. This is an extension to the earlier model provided for a typical monitor / collector exchange – just prior to requesting values for individual metric identifiers, an additional (optional) exchange is inserted. This allows the provision of a metric-specific filter.

In the cases studied to date, and those foreseen in the near future, it is expected that a string or short script will suffice for the filter. In this case, a regular expression (text string) was used, but in the case of a dynamic tracer like SystemTap or DTrace, a script in the respective language would be needed (also passed across the wire as text). This is identified as a requirement for any generic performance tools that wish to handle both sampled and trace metrics.



```
Terminal <9>
File Edit View Terminal Help
nathans@verge:~$ pmevent -x 'sshd' logger.perfile.syslog.records
host:      localhost
samples:   all
logger.perfile.syslog.records: 1 event records
  11:22:02.375 --- event record [0] flags 0x1 (point) ---
    logger.param_string "2011-07-31T11:22:02+10:00 verge sshd[14349]: Accepted p
ublickey for nathans from 192.168.0.101 port 55184 ssh2"
    logger.param_string "2011-07-31T11:22:02+10:00 verge sshd[14349]: pam_unix(s
shd:session): session opened for user nathans by (uid=0)"
    logger.param_string "2011-07-31T11:22:02+10:00 verge sshd[14349]: pam_limits
(sshd:session): Unknown kernel rlimit 'Max realtime timeout' ignored"
logger.perfile.syslog.records: 1 event records
  11:22:10.380 --- event record [0] flags 0x1 (point) ---
    logger.param_string "2011-07-31T11:22:09+10:00 verge sshd[14349]: pam_unix(s
shd:session): session closed for user nathans"
logger.perfile.syslog.records: 1 event records
  11:22:25.389 --- event record [0] flags 0x1 (point) ---
    logger.param_string "2011-07-31T11:22:24+10:00 verge sshd[14496]: Accepted p
ublickey for nathans from 127.0.0.1 port 54377 ssh2"
    logger.param_string "2011-07-31T11:22:24+10:00 verge sshd[14496]: pam_unix(s
shd:session): session opened for user nathans by (uid=0)"
```

Figure 15: Console Event Reporting and Filtering

Security-sensitive Parameters

One final observation can be made from this case study. It was found that additional levels of security were required for the event traces before the instrumentation could be deployed into the production environment.

The parameters of the individual events are so detailed that they contain potentially security-sensitive information, and with the distributed collector/monitor model PCP offers these parameters can be queried remotely. In this case study, system logs can be used to identify the access patterns of individual users, including user names, source system and system access method, details of projects they were working on and a daunting array of other low level information that could be used by an attacker in any number of ways. Not only information about users of the application are logged, but also the system administrators details and access patterns.

Contrast this to the simple, often numeric, values that are typical of sampled performance metrics. This is identified as a second requirement of any performance collector components that export event trace metrics – consideration must be paid to the security implications of doing this, and access controls are mandatory.

For this case study, extensions were made to *pmdallogger* to ensure the requesting client system is in an access controlled list of hosts that have been granted access to event traces from the collector host. A more complete solution to this issue would involve a user-based access control mechanism, such that access to individual metrics could be allowed or disallowed for different users. This is planned as future work, but for this specific study the simpler host-based model proved sufficient. Additionally, to prevent over-the-wire snooping, encryption is also planned as a future extension.

Data Warehouse Case Study

Servers from the production environment of a globally distributed SaaS company are recording data about many aspects of system performance around the clock. While invaluable for triaging and diagnosing individual issues as they arrive, the recorded data is not easily used for comparing some or all of the geographies to each other over long time periods.

A performance data warehouse has been implemented to provide technology decision makers with a powerful querying mechanism for this higher level information. The raw data covers physical hardware utilization for active servers, application internal metrics (request rates, response times), storage utilisation, network bandwidth consumption, and so on. The same performance metrics are available for a number of different datacenters, and with the added dimension of time, this forms a business intelligence “cube” depicted in **Figure 16**. The goal is to be able to quickly see long term trends for comparable services across data centres, summarising vast amounts of historical data offline for rapid online interrogation and reporting. Overall trends are relatively easily observed, and statistical outliers or otherwise unusual patterns of activities stand out when comparing like to like systems - as in **Figure 17**.

Metrics are collated from the fine-grained per-host daily logs, rate-conversion of counters is performed, per-minute values are calculated, and everything is fed into the data warehouse for off-line preparation.

This initial data extraction is a time-consuming process. It is performed weekly and outside of business hours. It forms the first phase of the ETL process (Extract, Transform, and Load) that is typical of a business intelligence system such as this.

In this case, the extraction phase involves reading log files from the past week gathered from machines in many different datacenters, and completes in around five hours of elapsed time. This import is an automated process, and insight into where the time is spent during extraction would be useful in determining the effects of growth in the data being imported.

This is used to plan for new datacentres coming online, and in seeking areas to optimise in the overall warehouse import process to ensure the batch load (Extraction) does not extend into working hours, and the processed (Transformed, Loaded) data is available as soon as possible.

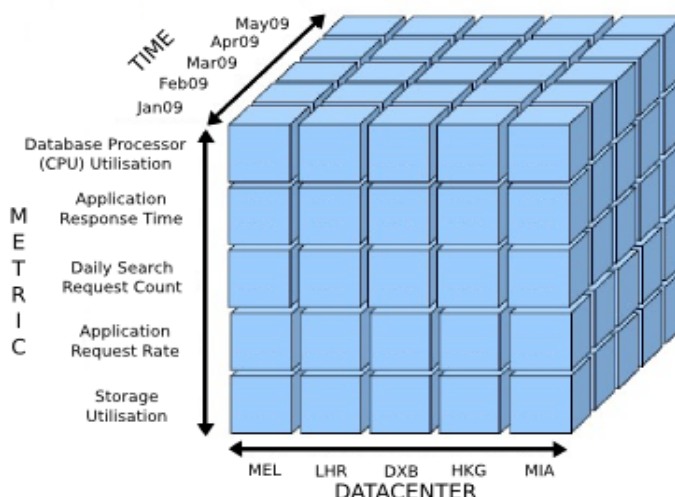


Figure 16: A Performance Cube

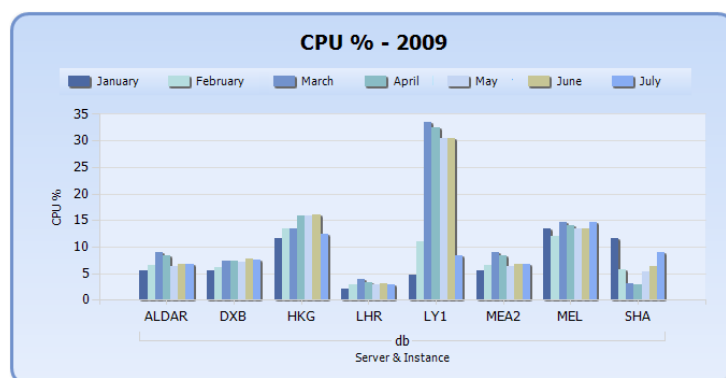


Figure 17: Sample Data Warehouse Report

Instrumenting the Extraction Phase

Scripts handle the exchange of data from the performance archives, into SQL statements. These are POSIX shell scripts which execute command line utilities to extract the log data and also to produce the SQL that inserts the data into the warehouse. The scripts run on a **logs** server, with local storage for the log files, and send data in the form of SQL statements to a remote **warehouse** server.

Both machines are running the Performance Co-Pilot software, sampling the regular array of system level and database-specific metrics suitable for these hosts on a relatively frequent interval (typically using ten second intervals). In addition, the extraction scripts have been instrumented such that each shell command executed is injected into the PCP framework using a new experimental PMDA – *pmdabash* – custom built for the purposes of this case study.

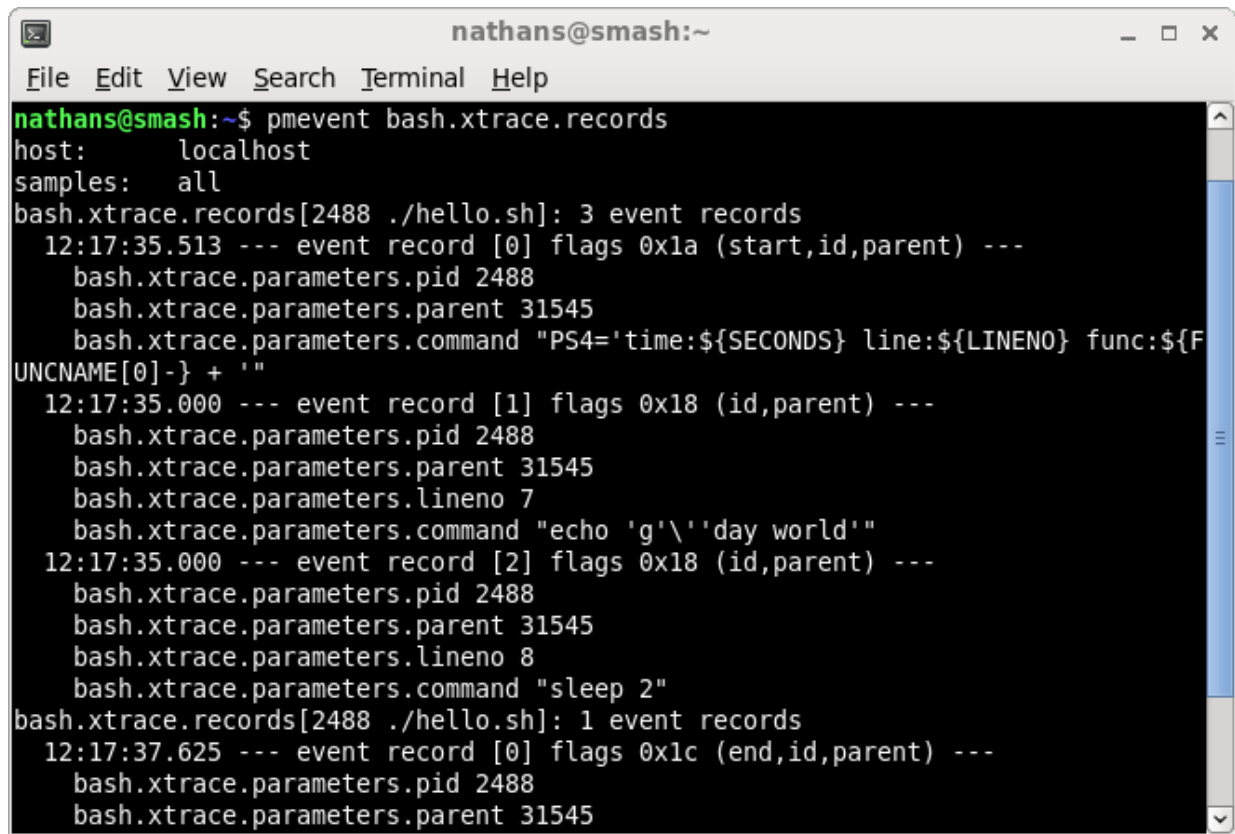
Using features of current versions of the *bash* shell, this co-opts the well-known “sh -x” tracing mechanism, but instead of reporting each executed command on the standard error stream of the shell as it is executed, these trace events are sent to a named pipe which the script initially creates.

The PCP *bash* PMDA running on the logs server detects when each new shell script is started, and begins reading events (command execution traces) from the named pipe until each shell completes. As described in the earlier methodology, these events are then available to PCP client tools for recording, reporting (as in **Figure 18**) and visualisation along with the sampled metrics.

bash.xtrace.records

For good measure we also track the total number of events observed and export this count as well as an additional PCP metric.

bash.xtrace.count

A screenshot of a terminal window titled 'nathans@smash:~'. The terminal shows the output of the command 'pmevent bash.xtrace.records'. The output displays event records for a script named './hello.sh'. The first record shows the start of the script at 12:17:35.513. Subsequent records show the execution of 'echo 'g\'\'day world\'\'' and 'sleep 2'. The final record shows the end of the script at 12:17:37.625. The terminal window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The background is black with green and white text.

```
nathans@smash:~$ pmevent bash.xtrace.records
host:      localhost
samples:   all
bash.xtrace.records[2488 ./hello.sh]: 3 event records
12:17:35.513 --- event record [0] flags 0x1a (start,id,parent) ---
  bash.xtrace.parameters.pid 2488
  bash.xtrace.parameters.parent 31545
  bash.xtrace.parameters.command "PS4='time:${SECONDS} line:${LINENO} func:${F
UNCNAME[0]} + '"
12:17:35.000 --- event record [1] flags 0x18 (id,parent) ---
  bash.xtrace.parameters.pid 2488
  bash.xtrace.parameters.parent 31545
  bash.xtrace.parameters.lineno 7
  bash.xtrace.parameters.command "echo 'g\'\'day world\'\'"
12:17:35.000 --- event record [2] flags 0x18 (id,parent) ---
  bash.xtrace.parameters.pid 2488
  bash.xtrace.parameters.parent 31545
  bash.xtrace.parameters.lineno 8
  bash.xtrace.parameters.command "sleep 2"
bash.xtrace.records[2488 ./hello.sh]: 1 event records
12:17:37.625 --- event record [0] flags 0x1c (end,id,parent) ---
  bash.xtrace.parameters.pid 2488
  bash.xtrace.parameters.parent 31545
```

Figure 18: Sample Script Instrumentation

For every command (event) executed by the instrumented shell scripts, a timestamp and several parameters giving details of the command are also extracted, as follows:

```
bash.xtrace.parameters.pid
bash.xtrace.parameters.parent
bash.xtrace.parameters.lineno
bash.xtrace.parameters.function
bash.xtrace.parameters.command
```

This is a particularly interesting instrumentation case study because of the richness of the event parameters. They provide parent-and-child relationships when one instrumented shell script executes another. They provide start-and-end pairs when a script first enables tracing until it exits. The nature of the import process is such that there are both short-lived and longer-running scripts, the latter executing the former as child processes. Also, the shell provides some additional event parameters of interest to analysts too – line numbers and shell functions, and of course the executed commands and their full command line.

This can be accomplished without modification to the underlying bash process, although the scripts need to be instrumented in order that the pipe be created and used. A trivial example follows, which produces the sample console shell trace shown earlier in **Figure 18**.

```
#!/bin/bash
. /etc/pcp.sh                # make instrumentation available

pcp_trace on $0 $@           # enable tracing to a named pipe
sleep 2
echo "g'day world"
exit 0
```

Table 2: Shell Event Trace Instrumentation

Process Visualisation

Instrumentation from the full data warehouse import execution has been analysed. The primary goal is to gain a deep understanding of where the elapsed time is spent during the import process, such that decisions can be made about where optimisations could be made, if indeed they should be made at all. These are both production systems with several different (concurrent) uses – we have not attempted to isolate these runs though, as this is unrealistic – we want to observe the system exactly as it operates in reality, which means some noise activity must be dealt with.

Starting with the high-level sampled operating system metrics related to processor, network, storage, and memory utilisation, the following initial observations have been made:

- Aggregate processor utilisation for both systems (database **warehouse** and **logs** import servers) has periods of saturation followed by idle times. Relatively constant background activity is observed from other, unrelated, work as well.
- Network traffic on both systems has short bursts of saturation followed by long idle periods. Some background traffic is also present, and is not constant like the processor activity but appears much more random.
- Both systems have effectively sized memory subsystems, and memory is acting as an effective cache for the entirety of the import run for both systems. Based on the observed sampled data we quickly discard further interest in memory as a potential performance bottleneck.
- The **logs** import storage subsystem shows some initial read-dominated disk I/O load while the data to be imported is read from stable storage. This is a relatively short period however, and once that data has been read into memory (buffered I/O), subsequent passes of the import show no additional I/O activity at all (consistent with the earlier memory utilisation observations). On the database **warehouse** server, at no time is disk I/O a limiting factor – SQL updates streaming from the network are written asynchronously, in an effective and efficient manner.

At this point, using sampled operating system data alone we have an good overall picture of performance and platform utilisation during the process, and the analysis focus shifts to using the additional trace data we have gathered.

An example of unified trace and sampled data graphs is shown in **Figure 19**. This shows activity on the logs server - network interface traffic and processor utilisation metrics (top and middle), and event traces from the import **bash** shell scripts in the final (bottom) graph. They share a common time axis, along the base.

A number of fascinating observations can be made from these graphs:

- Being able to visualise the child shell processes (the horizontal multi-coloured “step down” effect) from each other and their common parent (the flat yellow line, correlated using the shared time axis) is of significant value to the analysis. Further, the ability to select individual events and examine the event parameters (the commands being executed, and their arguments) associated with these child process events gives effective, rapid feedback as to which part of the import process is executing at the time of the selected event.
- In addition to this feedback around progression of the import process, we also receive effective correlation feedback from the sampled graphs as to which resources are most heavily utilised at any particular time point. We observe that the import process progresses in phases where we are CPU resource limited (scanning through raw data, producing SQL update statements), followed by an idle period of similar length, followed by the next CPU limited section. One might postulate that the network transfer correlates with the time we generate no new events (and associated lack of CPU utilisation). However, sampling the network traffic metrics shows us there is *no change* in traffic out at this time. The analysis proceeds to the warehouse server where the truth is revealed – we become CPU bound on the database during this time, and the driver script is blocked while the SQL update statements complete before continuing to the next phase.
- The green rectangular highlight area in **Figure 19** shows another interesting phenomenon. Clusters of events – more dense initially (this is due to several children scripts being started all at once, in parallel), then some finish quickly, and there are some stragglers. Correlating this trace data to the sample CPU utilisation data, we observe that the system is CPU bound initially, then with a slight tail (while stragglers finish), and then dropping off completely to background utilisation levels for a time (for reasons described in the second point above).
- As a generalisation of the previous two points, one can observe that both event clusters (many points in close time proximity) and the horizontal event begin-end pairing concept, provide highly effective visual correlations with the metrics from the sampled graphs.

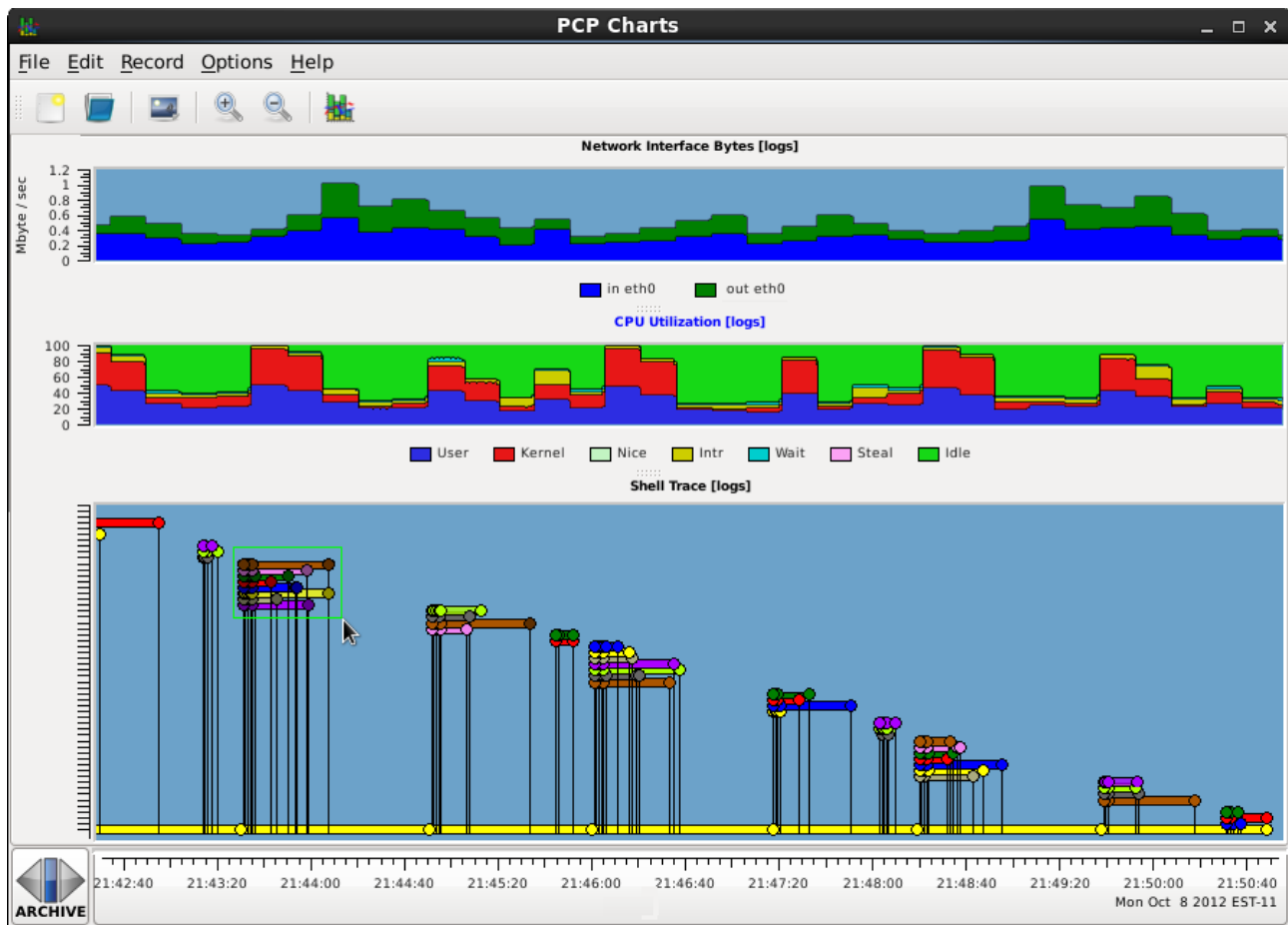


Figure 19: Warehouse Import Drill-Down

A number of potential areas for improvement have also been identified through use of the extended *pmchart* tool. When dealing with large numbers of events, improvements around interactive response time would be helpful. This might be achieved through a transparent overlap-culling process, hiding events with the same identifier and in close temporal proximity. Also, providing the user with a client-side filtering mechanism may be useful.

While demonstrably useful in its current form, the selection mechanism could also be improved further. In the situation where dense event clusters occur, as described earlier, the point selection mechanism is somewhat ineffective due to the close proximity of events. Additionally, when selecting groups of events in these clusters, the simple pop-up window quickly becomes overrun with detail which scrolls off the screen. A more sophisticated detail display window, with a mechanism for driving intelligent client-side filtering would be ideal.

Future work

Per-user security. The workaround put in place (discussed at the end of the first case study) is a stop-gap measure. A more comprehensive authentication model is needed, such that an individual user can be identified and allowed access to only that trace information that they should have access to. In addition, communicating trace data with sensitive event parameters over the wire warrants extensions to the over-the-wire protocol to at least optionally encrypt this information.

Validation through additional tracing agents. Extension into additional domains to further assess suitability of the agent-based model to event tracing - particularly the more common tracing frameworks (ETW, SystemTap, DTrace, LTTng) - would allow many more users and much more trace data to be injected into the system. This would provide further insight into whether the techniques described for unifying the two performance models are generally applicable.

Consideration for retrospective analysis and event filtering. The approach taken to filter events at the PMDA level, with domain-specific filtering, is a good approach allowing different tracing agents to be properly configured in the live mode of operation (e.g. with DTrace scripts, SystemTap scripts, ETW XML configuration, and so on - depending on the tracing domain). However, when performing analysis over historical data, this model no longer applies. This is highlighted in **Figure 20** (especially when compared to **Figure 9**).

The performance archives are such that they may be totally removed from the monitored system, and the contents analysed or replayed elsewhere. This suggests that a generic filtering mechanism suitable for the larger amounts of historical data would be useful. It could perhaps use the same user-facing mechanism as in live mode (the *pmevent -x* option, as in **Figure 15**) but a purpose-built language that allows filtering of any event metric would be required. Fortunately, this is made simpler by the fact that the metadata associated with all event parameters has been captured in the archive as well, but it would still be a significant area of further work well worth researching.

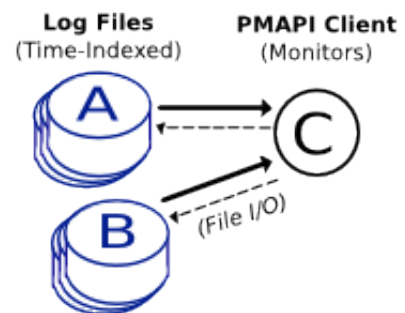


Figure 20: Retrospective PCP Analysis

Visualisation improvements. In the area of selection feedback for events, it would be worthwhile investigating a display mechanism that goes beyond the simple pop-up window prototyped here. A tabular display of the events and their parameters from an initial selection might allow subset selection, which could be fed back into the original event display (providing a visual client-side filtering model).

Layout of the new event trace graphs is another area where much improvement could be done – both in terms of positioning the elements, as well as rendering performance. Initial experimentation with large numbers of events shows that the simpler models begin to fall down as the number of visualised events increases.

Conclusion

Understanding and improving end-to-end performance of requests passing through a distributed application or other complex system, is a difficult undertaking. Information is available in the form of sampled metrics and event traces, and use of both forms at various stages of the analysis process has been demonstrated to be valuable.

In order to inform our research as to the best methods for merging the collection and presentation of these two sources of performance data, we have cast a critical eye over potential methods for mining this data – in both an automated fashion, so as to reduce or classify the data before presenting it, and also visually.

The research literature indicates that there is a clear and repeated need for combining these sources of data. Some have gone so far as to attempt to reconstruct sampled data from system event traces, which becomes unnecessary with a more flexible unified data model. Several very large scale computing environments are reporting excellent results using exclusively one or the other analysis technique. Providing a framework where both event trace and sampled data are extracted and presented side-by-side and on equal footing has proved an intriguing area of exploration.

In the early stages of investigation, fundamental differences in requirements around security were reported. These result from the fine-grained details encoded in event traces, which are not present with the generally coarse sampled data.

A separation between analysis (client/monitor) and data extraction (server/collector) is common in sampling based tools, and this separation can be used to implement live monitoring and retrospective monitoring transparently. For analysis tools that wish to use both sampled and traced metrics, event tracing presents an inherent need for these tools to be able to initially transfer filtering information to the extraction framework. This information is used to configure the tracing for the analysis tools on a per-user or per-session basis, allowing for multiple concurrent consumers of event traces (the traced event only happens once). By contrast, sampling is relatively cheap and multiple samples can be taken in quick succession by different users without interference with each other or the system under observation.

Exploration of a combined mechanism for visualising sampled and traced metrics has been undertaken. This presents direct visual correlation by aligning the charts around a shared time axis. A representation of the inherent structure that exists between some events has been trialled, demonstrating that significant value can be gained by associating identifiers with events at the time they are generated, and propagating that through the monitoring system to the analysis tools for display. Issues around display “zoom” levels have been discussed, and a tendency for event trace data to visually overlap has been found which exists to a far lesser extent with sampled data. Differences in the requirements of the interactive selection models used with each has also been discussed.

The postulation that a performance analyst gains from unifying event trace and sampled performance data has clearly been upheld. The ability to “drill down” into far greater levels of detail has been demonstrated to be highly effective.

Finally, the agent-based domain model that is commonplace in toolkits such as the Performance Co-Pilot, begins to show promise as a mechanism for combining trace data from multiple trace sources. Generic tools that can combine both sampled and traced performance data from arbitrary instrumentation sources have significant value.

References

- F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, M. Desnoyers (2011).
"Recovering System Metrics from Kernel Trace"
Proceedings of the Ottawa Linux Symposium 2011
- N. Scott (2011).
"Event Tracing: A Runtime Cost Analysis"
Proceedings of the Computer Measurement Group's 2011 International Conference
- R. Fonseca, M. Freedman, G. Porter (2010).
"Experiences with Tracing Causality in Networked Services"
7th USENIX Symposium on Networked Systems Design and Implementation
- B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, D. Beaver, S. Jaspan, C. Shanbhag (2010).
"Dapper, a Large-Scale Distributed Systems Tracing Infrastructure"
Google Technical Report
- M. Desnoyers (2009).
"Low-Impact Operating System Tracing"
Doctoral dissertation, École Polytechnique de Montréal
- R. Fonseca (2008).
"Improving Visibility of Distributed Systems through Execution Tracing"
Doctoral dissertation, University of California at Berkeley, Berkeley
- M. Bligh, M. Desnoyers, R. Schultz (2007).
"Linux Kernel Debugging on Google-sized clusters"
Proceedings of the Ottawa Linux Symposium 2007
- R. Fonseca, G. Porter, R. H. Katz, S. Shenker, I. Stoica (2007).
"X-Trace: A Pervasive Network Tracing Framework"
4th USENIX Symposium on Networked Systems Design & Implementation Proceedings
- I. Park (2006).
"Core System Event Analysis on Windows Vista"
Proceedings of the Computer Measurement Group's 2006 International Conference
- V. Prasad, W. Cohen, F. Ch. Eigler, M. Hunt, J. Keniston, B. Chen (2005).
"Locating System Problems Using Dynamic Instrumentation"
Proceedings of the Linux Symposium 2005
- M. Gaber, A. Zaslavsky, S. Krishnaswamy (2005).
"Mining Data Streams: A Review"
Association of Computing Machinery SIGMOD Record, 2005
- B. Cantrill, M. Shapiro, A. Leventhal (2004).
"Dynamic Instrumentation of Production Systems"
USENIX 2004, Annual Technical Conference, General Track Proceedings
- E. Metz, R. Lencevicius (2004).
"Performance Data Collection: Hybrid Approach"
Proceedings of the Workshop on Dynamic Analysis (WODA) 2004
- S. Teoh, K. Zhang, S. Tseng, K. Ma, S. Wu (2004).
"Combining Visual and Automated Data Mining for Near-Real-Time Anomaly Detection in BGP"
Proceedings of the 2004 ACM workshop on visualisation and data mining for computer security

- I. Park, R. Buch (2004).
"Event Tracing for Windows: Best Practices"
 Proceedings of the Computer Measurement Group's 2004 International Conference
- P. Barham, A. Donnelly, R. Isaacs, R. Mortier (2004).
"Using magpie for request extraction and workload modelling"
 Proceedings of the 6th conference of the Operating System Design & Implementation Symposium
- N. Gunther (2004).
"Analysing Computer System Performance with Perl::PDQ"
 Springer-Verlag, Berlin Heidelberg, ISBN:3540208658
- B. Tierney, D. Gunter (2003).
"NetLogger: A Toolkit for Distributed System Tuning and Debugging"
 Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management
- P. Barham, R. Isaacs, R. Mortier, D. Narayanan (2003).
"Magpie: online modelling and performance-aware systems"
 Proceedings of the 9th conference on Hot Topics in Operating Systems
- K. McDonell (1999).
"System Level Performance Analysis with Performance Co-Pilot"
 Conference of Australian Linux Users 1999
- R. Jain (1991).
"The Art of Computer Systems Performance Analysis"
 Proceedings of the Computer Measurement Group's 2001 International Conference
- C. E. Shannon (1949).
"Communication in the presence of noise"
 Proceedings of the Institute of Radio Engineers, Volume 37