End-to-end tracing considered essential

Dushyanth Narayanan (dnarayan@microsoft.com), Microsoft Research Cambridge

Abstract

Concurrency and complexity are major obstacles to understanding application performance in high-performance systems. We advocate end-to-end event tracing as the correct way to expose performance information for both human and automated analysis. We describe its advantages over traditional performance counter data, and illustrate its uses in performance visualization, prediction for capacity planning, and anomaly detection. We conclude with a look at opportunities and challenges presented by a widespread deployment of end-to-end tracing.

1 Motivation

System administrators are constantly challenged by the need to understand performance. When performance is unsatisfactory, is it a transient or a persistent problem? Is the solution a reconfiguration of existing hardware, or the purchase of new hardware? For a given budget, which new hardware would provide the most performance improvement? These questions are not easy to answer even for a skilled administrator, and they become ever more difficult as systems grow more complex and highly concurrent. Complexity increases the number of different components and subsystems that might impact performance; concurrency causes highly interleaved execution of transactions, making it difficult to analyze the performance of a single one in isolation.

Current systems provide little help to human administrators in understanding their performance, and none at all for automated resource and performance management. The state of the art consists in exposing a large number of *performance counters*: aggregated views of the utilization of physical or virtual resources [3, 5, 8]. Administrators identify problems by comparing these counters with some threshold values ("the disk queues are too long"). Such counters provide a narrow view of the system and do not identify the global bottlenecks ("if disk queues are long, should I buy more memory, faster disks, or reconfigure the database?"). Further, with 400+ performance counters, it is extremely difficult to know exactly which ones are relevant and what the correct thresholds are. Finally, aggregate counters do not offer any insights into response time, since they do not distinguish between background and foreground (critical-path) resource usage.

We argue that the correct approach to exposing performance data is *end-to-end tracing* of resource usage, rather than mere counting of aggregate resource usage or performance statistics.

End-to-end tracing is characterized by

1. **Cheap, fine-grained trace events** with high-precision timestamps.
2. **Tracing on fast/common paths**, not just anomalous ones.
3. **Precise resource accounting**: one event for each resource usage (disk I/O, buffer allocation, etc.), or in the case of execution resources suc as the CPU, for each context switch.
4. **Separating demand from service**: Highly concurrent systems multiplex a large number of requests onto a smaller number of execution contexts such as threads and processors. Also, a single request can move across threads, processes, and machines. By recording the execution context of each resource usage event, we separate the demand process — the resource usage of each transaction — from the service process — the scheduling of transactions on the underlying threads and processors.
5. **Tracing synchronization and control transfer**: One way to track request execution through the system is to maintain a global context or "request ID". This requires propagation of the request ID across software components and machines, requiring extensive changes to existing APIs and protocols. Instead, we augment events with local context such as thread ID, and also track synchronization or context switch events indicating transfer of control from one context to another (e.g. messages, work queues, RPCs, scheduler context switches). This avoids propagation of a global ID, and also lets us flexibly define "request" at different granularities: for example, an http request might generate several SQL "sub-requests". Tracing synchronization events is also useful for diagnosis of synchronization bottlenecks.
6. **Extracting in-request concurrency**: In addition to concurrency across independent requests, we might also have concurrency within a request due to overlap of processing, asynchronous I/O, and asynchronous RPCs. Tracking context switches and synchronization events allows us to extract this concurrency as a partial ordering.

End-to-end tracing provides several advantages over performance counters:

- *Disaggregated view*: performance and resource usage of individual transactions.
- *Control flow information*: ordering/concurrency/dependencies within a transaction.
- *Precise accounting*: each resource usage event is assigned to exactly one transaction.
- *Flexibility*: in addition to the original performance counters, new ones can easily be added by computing a different aggregate on the trace data (e.g. "how many transactions of type 'search catalogue' issued more then 10 disk read requests?")
- *Response time analysis*: whereas performance counters only identify throughput bottlenecks, end-to-end tracing tracks per-transaction resource usage as well as ordering dependencies, which allows measurement and prediction of execution latency.

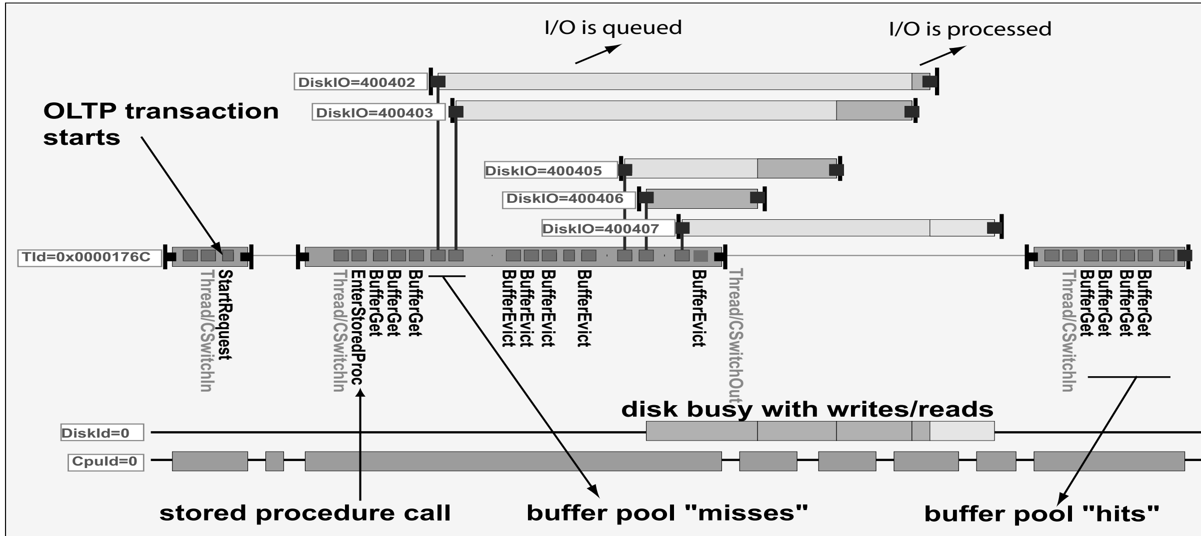


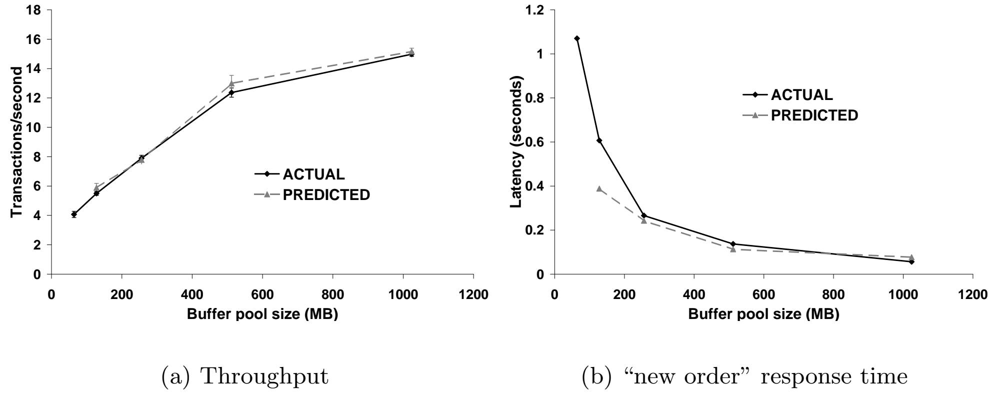
Figure 1: Timeline view of a single OLTP transaction

2 Status

As part of the Magpie project at MSR Cambridge, we have successfully applied end-to-end tracing to several scenarios, including that of a two-tier web service [1, 2, 4, 7]. We have instrumented Microsoft IIS Server and SQL Server to trace resource usage and control flow, and used the resulting traces as input to a variety of applications. This section briefly describes our tracing infrastructure and and three of these applications.

Low-overhead, non-blocking event tracing is provided by Event Tracing for Windows (ETW) [6]. Events are posted by calling a Win32 API function with an event type and data fields specific to that event type. ETW timestamps each event with the processor cycle counter and asynchronously flushes events in timestamp order to disk or to a consuming process. All critical-path operations use CPU-local data structures to enable multiprocessor scaling, and we are optimistic that the overheads will scale well with system size. Our measurements of SQL Server running a TPC-C workload on a single processor show an average of 500 events/transaction, 1000 cycles/event, and 68 bytes/event of trace data. Extrapolating this to the fastest TPC-C system as of date (3,210,540 tpmC with 64 processors at 1.9 GHz) [9], we get a CPU overhead of 5% and a trace data rate of 8 MB/s. This could be reduced further through careful optimization. More importantly, overhead can be reduced arbitrarily through sampling, i.e. selective enabling of events at runtime.

Our first application was to visualize end-to-end performance in a way that traditional performance counters do not allow. As we can extract the control flow and resource usage of individual transactions, we are able to generate "timeline" views such as Figure 1. Such views allow us to



(a) Throughput

(b) "new order" response time

Figure 2: Predicting TPC-C performance when memory size is doubled

examine in detail the execution of a single transaction, although in reality it was heavily interleaved with many others. Traditional performance counters (e.g. "CPU load") can also be easily computed from the trace data, with the benefit that we can dynamically add new performance counters without any reconfiguration of the live system.

A second application is automated performance prediction for capacity planning. By measuring the resource demand of real applications on live systems, end-to-end tracing lets us accurately answer "what-if" questions about hypothetical new hardware with *quantitative* predictions of the expected change in performance [7]. Answering such "what-if" questions automatically is of great benefit to system administrators in making informed and cost-efficient upgrade decisions, but today's systems lack such a self-predictive capability. Figure 2 shows graphically the answer to one such "what-if" question: "what will happen to throughput and response time of my current workload if I double memory size". These predictions were generated by applying parametrized models of SQL Server's CPU, disk, and buffer management to the per-transaction resource demands obtained by tracing a live system.

A third promising use of end-to-end tracing is *anomaly detection*. Given detailed per-transaction information, we could construct models of "normal" transaction resource usage, and automatically identify abnormally behaving transactions without requiring any knowledge of the application code or semantics. Our results show that simple clustering-based models can differentiate between transaction behaviours based solely on their resource usage traces [1], and we are hopeful that this can be developed into a full-fledged outlier detection mechanism.

3 Agenda

The main barrier to widespread use of end-to-end tracing is not the overhead or the lack of applications but the absence of appropriate instrumentation in today's commercial DBMS. Tracing is used for ad-hoc debugging of rare or erroneous code paths, but there is no systematic effort to trace the common or fast path for a complete picture of end-to-end performance. We recommend strongly that designers and developers of DBMS components include end-to-end tracing as a high-level goal from the start. Specifically, they should aggressively instrument all code paths with events (which can be selectively disabled at runtime), especially all physical and virtual resource usage, synchronization, control transfer, and context switching.

Our current research agenda is to improve the applications of end-to-end tracing that we described here: to develop new ways to query and visualize trace data; better models for performance prediction and capacity planning; and models of transaction resource demand for outlier/anomaly detection. No doubt, widespread use of end-to-end tracing will create new applications, with associated challenges in distribution collection and analysis, storage, and management of trace data.

References

[1] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[3] IBM. DB2 performance expert. `http://www-306.ibm.com/software/data/db2imstools/db2tools/db2pe/`, Mar. 2005.

[4] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[5] Microsoft. Improving SQL Server performance. `http://msdn.microsoft.com/library/en-us/dnpag/html/scalenetchapt14.asp`, May 2004.

[6] Microsoft. Event Tracing for Windows (ETW). `http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp`, Feb. 2005.

[7] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. `http://www.research.microsoft.com/~dnarayan/dbperf.pdf`, Mar. 2005. *Submitted for publication.*

[8] Oracle. Oracle database manageability. `http://www.oracle.com/technology/products/manageability/`, Jan. 2005.

[9] Transaction Processing Performance Council. Top ten TPC-C by performance. `http://www.tpc.org/tpcc/results/tpcc_perf_results.asp`, Mar. 2005.