Analyzing Application Performance Using a System Monitoring Database

Aaron Brown, David Oppenheimer
{abrown,davidopp}@cs.berkeley.edu

Abstract

In this paper we introduce a new approach to system performance analysis. Based on the use of a system monitoring database, SPADE (System Performance Analysis Database Engine) overcomes many of the shortcomings of traditional performance analysis tools. SPADE collects monitoring data from all levels of the system, including the application, and stores it in a central repository backed by a relational DBMS; the granularity of the system instrumentation is automatically matched to the application's notion of a request, allowing the database to correlate all monitoring data with semantically-meaningful application requests. We demonstrate that the SQL-based query facility of the database enables unprecedented levels of flexibility, power, and ease-of-use in analyzing the data: a user of our system can write simple queries to examine the collected monitoring data at multiple levels of detail, to locate system bottlenecks and unusual behavior, and to easily test hypotheses about the system's performance, all without having to write ad-hoc tools or perform multiple experiment runs.

1 Introduction

Networked servers running data-intensive applications such as web services and databases are playing an increasingly important role in modern computing. It has been recently argued that these "infrastructure" systems will form the most significant class of large computer installations as we move into the next era of computing, the "Post-PC era" [Hen99, Pat99]. One of the most critical qualities of data-intensive infrastructure servers is their per-request performance (*i.e.*, response time), as this metric translates directly into user-perceived performance. To provide the best user experience, it is important that these servers be highly tuned to achieve the best performance possible. This tuning in turn requires an understanding of the underlying reasons why the system performs as it does and, in particular, how various system characteristics impact the server application's perceived performance.

However, most of today's data-intensive servers are extraordinarily complex and exhibit performance characteristics that do not lend themselves to easy analysis. One of the more significant difficulties is that system performance is often governed by the interactions amongst many different parts of the system, from hardware devices (such as disks) to the operating system to the application itself. Traditional performance monitoring tools are not up to the task of collecting performance data from all of these different subsystems, correlating that data with application performance, and integrating it to present a unified view of the performance of all the system internals as well as of the application. For example, the performance tools built into most operating systems (such as *systat*, *vmstat*, *netstat*, *top*, and so forth) provide a wealth of detail about the internal operation of the operating system, but do so only on a very coarse, fixed-timestep granularity. As a result, they provide no way to correlate that data with the performance of any given application-level service request; at best these tools can provide limited insight into the aggregate throughput of an application.

We believe that the solution to this predicament lies in the application of database technology to the problem of analyzing and understanding system performance. A relational database provides a natural repository through which diverse performance and monitoring data collected from all levels of the system can be centralized and integrated. It also provides a powerful query mechanism that can be used to investigate the data, to test hypotheses about the impact of various components on system performance, and to perform data reduction and summarization. Advanced database systems also provide the ability to

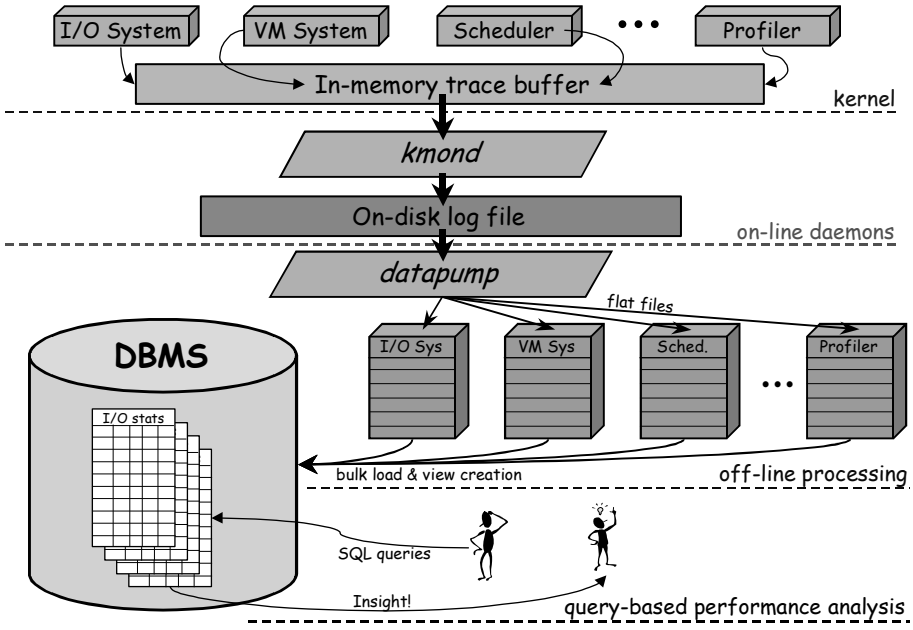

Figure 1: SPADE System Overview

perform data mining and on-line analytical processing (OLAP) operations that can potentially extract novel and unexpected correlations in the data, providing new insight into system performance.

In this paper, we describe the architecture and implementation of SPADE (System Performance Analysis Database Engine), a system that demonstrates the benefits of using a relational database to store and analyze system monitoring data. We have fully instrumented both a single-node operating system kernel and a web server application to collect performance data and to insert it into a relational system monitoring database; the application and kernel instrumentation are described in detail in Sections 3 and 4, respectively, while Section 5 describes the process by which the monitoring data is placed into the database. In Section 6, we show how the database's query capabilities can be used to transform the raw monitoring data into a format that correlates subsystem performance statistics with per-request application response time. Section 7 illustrates the power of using a system monitoring database by demonstrating how database queries can be used to identify regimes of poor performance and to trace the causes of that performance degradation, potentially providing the insight necessary to tune the system's hardware and software configuration. Our prototype implementation carries out the bulk of its processing and analysis off-line; Section 8 examines the performance implications of moving to an online model, and considers the suitability of various DBMSs for this application. In Section 9 we present some directions for future research, considering in particular how the use of an on-line system monitoring database could provide the foundation for automatic application adaptation to system performance bottlenecks. Finally, we present related work in Section 10, and in Section 11 we conclude.

2 System Overview

2.1 System structure

The performance data collection and analysis system that we have implemented consists of several components roughly partitioned into three groups: in-kernel instrumentation and data collection code, off-line data transformation tools, and the relational database used to integrate the monitoring data collected

and processed by the first two groups of components. A schematic overview of the system is presented in Figure 1.

The in-kernel instrumentation and data collection code serves two functions. First, it performs the actual OS- and device-level monitoring: small snippets of code spread throughout the OS kernel monitor and record system and device *events* as they occur and gather *snapshots* of summary statistics describing subsystem utilization, memory utilization, and so forth. The details of this instrumentation code are described further in Section 4. The other function of the in-kernel portion of our system is to provide a unified interface to the kernel's monitoring functionality. We provide a set of procedures that allows the various pieces of monitoring code described above to store their collected data as records within a large circular trace buffer allocated within the kernel's memory region; all records have a standard header, are automatically timestamped, and are tagged by type. The system also provides several system calls that allow a user process to extract the data from the buffer and to enable or disable each individual monitoring function. Synchronization of buffer access between the user process and the kernel is handled automatically by the centralized interface and is thus hidden from the code that actually performs monitoring or statistics gathering, keeping that code simple and fast.

The second group of components consists of tools that extract the monitoring data from the kernel buffer, perform some simple preprocessing and transformation of that data, and convert it into a format that can be directly bulk-loaded into a relational database. Our prototype system relies on three separate programs for this task (although they could easily be combined, and should be in a production-caliber implementation): the first, *kmond*, extracts the data from the kernel; the second, *datapump*, performs some simple computation and transformation of the data; and the third, *db-insert*, converts the output of *datapump* into the appropriate bulk-loading format for the particular database being used. More details on each of these tools can be found in Section 5.

The last component of the system is the database itself. Any relational database can be used for this component of the system, although we found that an industrial-strength commercial database was required to handle the volume and complexity of the monitoring dataset. Flexible query support is also an important requirement. We describe our experience with several different DBMSs in Section 8.

2.2 User experience

Analyzing a system's performance using the monitoring and analysis system that we have just described is a multi-step process with our prototype implementation. First, the user must instrument the application as described in Section 3 to inform the kernel monitoring subsystem of application-level events and to log application-specific performance metrics. Next, the user enables the monitoring subsystem and runs the application, subjecting it to the workload under which the user wants to understand the system's performance. If the workload in question is one that results in poor performance, the user can optionally monitor a second application run in which a "normal" workload is used to obtain a baseline characterization of the system to use as a comparison point. After collecting the traces from these application runs, the user invokes the off-line data processing tool chain in order to transform the data and load it into the database. At this point, the user can interact with just the database, issuing queries and examining the query results in order to investigate the system's performance at different points in the application workload, to test hypotheses about why performance might be slow, and to search for novel correlations between system and application performance. The key point here is that the data collection and analysis are completely divorced by the use of the database: the user does not need to decide on hypotheses or questions to ask before collecting the data, and at no point should the user need to perform additional tracing or application runs in order to pose and answer a different set of questions. Instead, the user can simply issue a new set of queries that aggregates and summarizes the collected data in a different manner.

The fact that so many steps are required in the analysis and data-collection process is primarily an artifact of our prototype implementation; we purposefully kept the various components and tools separate to allow us to change their implementations during development of the system. There is no inherent reason why the analysis process need be broken down into so many steps, and in fact a production implementation of our system would most likely combine tracing with on-line data transformation and database loading, eliminating all steps but the initial instrumentation and the final data analysis. We consider some of the issues in moving to such an online model in Section 8, below.

3 Application Instrumentation

Because SPADE's focus is on correlating application performance to underlying system behavior, the application to be analyzed must be instrumented to collect data about relevant application-level events. We therefore instrumented two applications, the Apache 1.3.3 web server and the PostgreSQL 6.4.2 database. Because we were interested in user-visible application latency, we recorded the response time measured between the user request being issued to the system (received by the back-end in PostgreSQL, received by the web server in Apache) and the response being returned to the user (returned from the back-end to the front-end in PostgreSQL, returned from the server in Apache). We also recorded the identity of the request, namely the web page requested in the case of Apache and the database query issued in the case of PostgreSQL. Because we examined only Apache in our application performance study, we describe the issues related to Apache's instrumentation here. The required instrumentation is similar in PostgreSQL.

There are actually two response time metrics of possible interest in Apache. One is the latency between the server receiving the request for a web page and the server returning the web page to the client; the other is the latency between the server receiving the request for a web page and the server closing the TCP connection with the client (after returning that page). While the extra time overhead of closing the TCP connection is of some interest, we are primarily interested in the code path from request receipt to page delivery, want the path related to the connection control aspects of the TCP protocol; moreover, we wanted to exclude as much as possible the impact of random variations in network performance which can have a significant impact on the overhead of closing a TCP connection. Thus while we measured the latency both including and excluding this connection-closing overhead, our performance study of the application used the latency measure that excludes the time spent closing the connection.

Because it is logically multithreaded (depending on the host platform, a separate process may be used for each logical thread, as is the case on our NetBSD-based system), Apache introduces a slight complication in mapping operating system statistics to application request units (*i.e.*, single requests for a web page). Because of the overhead of recording summary snapshots, it is desirable to record them only when necessary, *i.e.*, at the beginning and end of each HTTP request. The difference between the two snapshots, which is attributable to the query running during that time interval, can be computed as a postprocessing step. But the multithreaded nature of Apache allows multiple requests to be simultaneously serviced; this means that multiple summary snapshots may be recorded between the begin and end records inserted by the application for a single request. To allow easy aggregation of all the statistics collected during the execution of a query, we divide time into "epochs." Initiation or completion of server processing of an HTTP request defines the an epoch's boundaries; such a boundary is marked by the application making a system call that records a "begin request" or "end request" log record. An epoch is therefore the finest granularity on which statistics must be collected (and events timestamped) in order for the statistics relevant to a particular HTTP request to be computed. The epoch number can be thought of as a logical timestamp value, and the statistics collected during all epochs during which a query was running are aggregated to produce the system utilization information for that query. We will see that our

use of a database table mapping HTTP requests to epochs, combined with our storing summary snapshots and events in the database tagged by the epoch during which they were collected, allows us to easily aggregate the data relevant to any particular query.

To support application instrumentation we added an OS system call, `kmon_increment_epoch()`, that takes an application-defined type field and an application-defined log record structure. It increments the "current epoch" counter and causes summary snapshots to be logged into the trace buffer along with a "new epoch" log record that includes the current time and the application-defined log record data. In the case of Apache the type indicates whether the call was made to mark the beginning or ending of the servicing of a request, and the application-defined data provides the ASCII representation of the requested URL. In the case of PostgreSQL the application-defined data is the ASCII representation of the SQL query handled. Given this system call, instrumenting an application is very easy—`kmon_increment_epoch()` is simply called at the beginning and end of handling a request.

4 Operating System Instrumentation

Because we want to collect as many operating system statistics and events as possible, SPADE required significant operating system instrumentation. We modified the NetBSD 1.3.3 kernel to collect summary statistics once per epoch (*i.e.*, once each time the application calls `kmon_increment_epoch()`) and to continuously collect dynamic event traces; both types of records are tagged with the epoch during which they were collected as well as the hardware cycle counter timestamp at the moment the record is written. In this section we describe the statistics SPADE collects as summary snapshots and the events it records.

Summary snapshots were taken primarily from five kernel subsystems: the network stack, the file system, disk I/O handlers, the virtual memory system, and the interrupt handler. Some of these statistics, *e.g.*, those related to the network stack, were already collected by the NetBSD kernel, in which case we simply had to find and copy the relevant statistics into the in-memory trace buffer. Other information was computed by traversing kernel data structures. Determining the number of free buffer cache buffers of each size, the number of processes in each scheduler priority queue, and the number and type of active memory allocations required this type of direct examination of kernel state. Finally, kernel profiling data was also collected during each epoch; since a snapshot of the profiling buffer was captured at the end of each epoch, we classify this data as a summary statistic. In addition to the summary statistics, four types of events were recorded, corresponding to the occurrence of a SCSI error, a disk read or write, a process being descheduled on a context switch, or a system call being made.

The operating system instrumentation framework was designed with extensibility in mind. To this end, the support code needed by the instrumentation functions for kernel-level locking, memory management, and synchronization with the user-level daemon process (which writes log records to disk when the circular buffer of log records described in Section 2 becomes more full than a user-specified threshold) are handled by a single function, `kmon_allocate_record()`. `Kmon_allocate_record()` also records the epoch number and time (read from the hardware cycle counter) during which it was called. Thus instrumentation functions merely need to call `kmon_allocate_record()` specifying the size of the log record they wish to write, after which they write their data into the memory region returned. Of course additional locking may be necessary if the instrumentation functions themselves examine volatile kernel data structures.

4.1 Summary snapshots

A trace record corresponding to a summary snapshot is recorded whenever an epoch boundary is indicated by an application call to `kmon_increment_epoch()`. To maximize efficiency, most snapshot data is recorded as absolute values, with per-epoch differences (*i.e.*, deltas from the previous epoch) computed offline in the *datapump* program described in Section 5. The snapshot recording mechanism was designed with extensibility in mind; adding a new function to record snapshot data merely requires writing the function and registering it in a table of "callout" function pointers, each of which is called on an epoch transition.

Nine types of summary snapshot data were collected. *Network* statistics include information from all levels of the protocol stack, *i.e.*, TCP, UDP, IP, ICMP, and IGMP. The TCP statistics contain data about connections (number initiated, accepted, established, dropped, and closed), packets (control, data, and total packets and bytes sent, received, dropped, and retransmitted), packets received containing errors or that were received out-of-order, and so on. UDP statistics record the number of packets sent and received as well as the frequency of various error conditions. The IP statistics similar to those for UDP, except that they also include information about fragmentation and routing events. ICMP and IGMP statistics are of less interest since ICMP is used primarily for network monitoring and IGMP is only used by hosts acting as routers (which ours was not). *Disk* statistics include information about the number of transfers and total bytes transferred during each epoch, as well as the amount of time the disk was "busy" during the epoch. *Buffer queue* statistics include the number of free file system buffer cache buffers of each type and size. *Scheduler* statistics record the number of processes in each state (idle, runnable, sleeping, stopped, or zombie) and the number of processes in each of the system's 32 priority queues. *Virtual memory* statistics include information about the system's overall memory usage, including the amount of in-use and available real memory, virtual memory, and shared memory, as well counts of the number of jobs blocked due to paging, swapping, or disk I/O. *Interrupt* statistics count the number of interrupts that occurred at each interrupt priority during the epoch. *Summary* statistics consolidate a wealth of data from various kernel subsystems (focusing mostly on the VM system), including counts of context switches, traps, faults, system calls, VM cache lookups, copy-on-write operations, and page-allocation operations; information on the page-replacement algorithm (such as revolutions of the 2-handed clock); and statistics on the number and type of active and inactive pages.

The NetBSD kernel includes a tagged memory allocator that associates a type with every call to the in-kernel `malloc()` function. This allows us to extract *memory* statistics that provide a breakdown of operating system memory allocations by type/use, as well as by size. These statistics also include details on the request rate for each type and size of memory allocation, and flags that indicate whether processes blocked trying to acquire a certain type of memory.

Finally, we modified the NetBSD kernel to periodically sample the value of the program counter (via a routine called from the clock interrupt). Each sampled PC value is histogrammed to produce a running profile of where the kernel spends its execution time. At the end of every epoch, the histogram buffer is dumped into the monitoring log buffer as a summary statistic; the offline processing tools described below in Section 5 map the PC values in this histogram to kernel functions in order to provide a traditional flat profile of the kernel on a per-epoch granularity.

4.2 Events

In addition to periodic summary snapshots, SPADE generates a log record each time certain events occur. A *disk transfer* event occurs whenever a disk request is issued; the event is actually written to the log when the request completes so that the response time can be recorded, along with the size, logical and

physical block numbers, and read/write status of the transfer. A *SCSI error event* is recorded whenever the SCSI disk driver indicates an error was returned from the device; this event contains the error code returned. A *scheduler event* is recorded whenever a process is descheduled on a context switch; this event logs information available from the process control block of the process, such as the process ID, the parent process ID, the process owner's UID, the amount of time the process was running on the CPU during its most recent scheduling quantum, the process's priority and nice values, the user and system time consumed during its most recent quantum, the size of the process's text, data, and stack segments, and the number of page faults, block I/O operations, messages sent and received, and signals received during its most recent quantum. Finally, a *system call event* occurs each time a user process initiates a system call. For each system call, the kernel instrumentation code records the system call number, the process ID of the process initiating the call, the arguments to the system call, the return value and `errno` set by the call, and the elapsed time of the system call (in CPU cycles).

4.3 Instrumentation overhead

Unfortunately our kernel and application instrumentation resulted in significant overhead compared to an uninstrumented kernel and application. A workload of 1000 HTTP requests took about 19.1 ms/query with monitoring enabled, but only about 5.8 ms/query without monitoring, for a monitoring overhead of almost 300%. One obvious way to reduce this overhead is to use sampling, *i.e.*, to record statistics for only a fraction of the incoming HTTP requests. SPADE provides a system call to turn monitoring on and off on a per-record-type granularity to allow this type of sampling. Due to time constraints we did not investigate the accuracy-overhead tradeoff for sampling in the context of Apache running with SPADE.

5 Off-line Data Transformation and Loading

In this section, we describe the process by which the monitoring data described in the previous sections is extracted from the kernel and transformed into a format in which it can be easily bulk-loaded into a relational database. This process is accomplished via a set of tools, each of which we will describe in turn.

When monitoring data is collected by the kernel, it is placed into a large in-memory circular buffer, as described above. Thus, the first step in the data transformation and loading process is to extract the monitoring data from that kernel buffer and write it to a flat file. This procedure is handled by the kernel monitoring daemon *kmond*. *Kmond* is not an entirely off-line processing tool, as it must handle the complication that the kernel data buffer might fill up and thus must be periodically drained as the system continually collects monitoring data. Thus, *kmond* is structured as a daemon process that runs concurrently with the monitored application/OS system, and that interacts with the running monitoring subsystem via a series of system calls that give it synchronized access to the monitoring buffer. *Kmond* periodically wakes up (at a user-specified interval, by default 60 seconds) and executes the `kmon_sync()` system call. This system call momentarily quiesces the kernel's instrumentation subsystem and records the current beginning and end offsets of the active data region within the circular buffer. It then re-enables instrumentation and returns the two offsets to *kmond*. If the amount of active data in the buffer falls above a user-specified threshold, *kmond* memory-maps the kernel buffer into its address space and appends the portion that was active at the time of the sync to an on-disk log file. It then calls the `kmon_buf_reset()` system call, which resets the kernel's notion of the start of the active data area in the buffer to one record past the offset of the last record returned during the earlier call to `kmon_sync()`. Notice that during the time between the calls to `kmon_sync()` and `kmon_buf_reset()`, the kernel is still able to collect and log monitoring data. To reduce system perturbation, *kmond* does no further processing on the (binary) data it extracts from the kernel buffer. We note that during the experiments discussed in this paper, the buffer never filled up completely and thus *kmond* did not perturb the data collected.

Once the monitoring data file has been collected by *kmond*, it must be processed into a format suitable for loading into a database. In particular, the type of each data record collected must be identified, any needed processing of the data (to compute aggregates, differences, etc.) must be performed, and the data must be output in a delimited ASCII format readable by database loading tools. These tasks fall to the next program in our data-transformation tool chain, *datapump*. *Datapump* sequentially scans the trace file output by *kmond*, isolating each record in the trace and handing it to a type-specific function for further processing. All records begin with an initial processing step that extracts the epoch number and timestamp (expressed as a 64-bit cycle counter value) from the record, converts the timestamp from cycles to seconds, and computes from the timestamp a unique pair of signed 32-bit integers that can be used as a key for databases (like PostgreSQL) that do not support 64-bit primary keys. The type of processing that occurs next is dependent on the type of the monitoring record. For most event records (that is, records that indicate the occurrence of a dynamic event like a system call or an I/O operation), the logged binary data is simply converted to an ASCII representation and output in delimited form along with the epoch number, timestamp, type, and unique keys. A typical event record (in this case, for an `open()` system call) looks like this in the *datapump* output:

```
5|0.0635|0|22219975|KMON_SYSCALL|open|5|273|596412|0|438|0|4|0|0.0090
```

For most snapshot records (that is, records that contain counters and statistics about the current state of the system), *datapump* applies a transformation that converts two snapshot records, one at the beginning of the epoch and one at the end, into a single output record containing the differences between the snapshot values at the beginning and end of the epoch. For example, one particular snapshot record contains a field that holds the number of context switches that have occurred since the system was booted. The kernel instrumentation subsystem takes a snapshot of this field at the beginning and end of every epoch, and *datapump* uses those pairs of records to compute the number of context switches that occurred during the epoch. The differences are taken on the appropriate fields, and these numbers are converted to ASCII and output in delimited form similar to the example above, along with the epoch number, timestamp, type, and unique keys. Note that it would be possible to compute most of these differences within the kernel rather than relying on an external program like *datapump*; we chose to separate the functionality in our prototype implementation to give us more flexibility and to reduce the overhead of our instrumentation code, although a production implementation would most likely move the differences inside the kernel, if for no other reason than to save space in the monitoring data buffer.

Finally, there are a few record types that *datapump* handles specially. The most interesting of these is the program-counter history record type. As described in Section 4, the kernel instrumentation samples the program counter (PC) value periodically during every epoch, accumulating a histogram of PC values, in a way similar to traditional profiling tools like gprof [GKM82]. *Datapump* takes this histogram and maps it into a per-function execution time profile. It does this by extracting the symbol table from the OS kernel and mapping each profiled PC value to the function that contains it. After this mapping process is done, *datapump* computes the fraction of execution time spent in each function by dividing the number of samples in each function by the total number of samples taken. Any functions with a non-zero fraction of execution time are output as records in the standard delimited format along with epoch number, time, etc.

The output of *datapump* consists of a single flat file with one line per monitoring record/statistic gathered by the kernel instrumentation code. Each line is tagged by the type of monitoring record that generated it. The lines in the *datapump* output correspond directly to rows in the database tables, one table per record type. Since the data is already in delimited ASCII format, it is essentially ready to bulk-load into any relational database. In order to do this, however, a bit more off-line processing is required. The bulk-load facilities of most databases require that the data for each table be in a separate file, and

some require special syntax in the bulk load files (for example, PostgreSQL does not have a bulk load utility, but rather requires that the data file be wrapped with a SQL command and fed to the SQL interpreter). To transform the *datapump* output into the appropriate form, we feed it to *db-insert*, a Perl script that sorts the lines from the single *datapump* output file into separate files, one for each record type. It also adds any special syntax needed by the particular database that is the target for the bulk load. Note again that it would be possible to consolidate the tool-chain by moving *db-insert*'s functionality into *datapump*; we kept it separate in our prototype to make it easy to experiment with different relational databases requiring different bulk-load data file formats.

Once *db-insert* has been run on the *datapump* output, a set of bulk-loadable files, one per database table, is produced. At this point, the data is ready to be directly loaded into the database using the appropriate bulk-load facility, for example *bcp.exe* under Microsoft SQL Server. Before the bulk load can be performed, however the database schema must be created, as described in the next section.

6 Database Schema and Views

As described in Section 5, *datapump* and *db-insert* together load each trace record into a flat file corresponding to its record type in preparation for bulk loading into the database. In this section we discuss the schema that we used for our system monitoring database and the materialized views[1] we created over that database to simplify the writing of queries that extract from the database insight about performance trends and correlations.

6.1 Monitoring database schema

One database table was created for each log record type. The full schema appears in Appendix A.

Our goal in designing the database schema was to balance adherence to the relational model with a desire for storage efficiency and computational efficiency of queries. A wide spectrum of schema design options was available, ranging from the use of one table, indexed by epoch and timestamp, to store all statistics and events, to the use of one table, also indexed by epoch and timestamp, for *each* statistic and event-related datum collected (*i.e*,. each data member of a log record). We saw this problem space as two-dimensional: one dimension is a choice of whether to use a separate table for each statistic type (corresponding to a single log record type), and the other is whether to subdivide any log records (sets of related data gathered at the same time) so that each log record maps into multiple rows, with some subset of the log record's statistics in each row.

Since the instrumentation code writes a separate record to the log for each statistic type, and an online version of SPADE would insert log records directly into the monitoring database as they are created, we decided to use one table per statistic type (then each log record insertion would correspond to a single database INSERT operation). Even bulk loading is somewhat easier under this schema, since the bulk-load files can be generated by a single pass through the log, with each log record written to the bulk-load file corresponding to its type. A single table for all statistics would require concatenating data from all log records written during a single epoch before insertion, in the case of either bulk loading or direct insertion. The only drawback of the one-table-per-type schema is that examining statistics across multiple tables requires a join of those tables. Our hope is that a good query optimizer can make these joins almost as efficient as the projection of columns that would be required to answer a query to a single large table.

[1] Note that what we refer to as views in this paper are not views in the traditional sense, in that they are not updated automatically when their base tables are updated. They are instead simply tables that contain a transformed version of the data in the base tables.

We now briefly discuss the second axis, namely whether to subdivide any log record so that each one maps into multiple table rows. In this case some subset of the log record's statistics would appear in each row, along with some identifier of the log record from which the row came (to tie together rows from the same log record). A single row per log record minimizes the number of insert operations that need to be performed and minimizes the amount of space needed for the table (since epoch number and timestamp need to be stored once per row). The drawback to this approach, however, is that it makes computing aggregates across columns awkward since the aggregation operation and its column arguments then have to be written explicitly in the SELECT statement. For example, in constructing the KMON_SS_INQ table, we could have used one column for each scheduler priority queue. Then computing the number of processes in, say, the first ten queues during the first epoch would have required a SQL query of the form

```
SELECT number0 + number1 + number2 + number3 + number4 + number5 +
       number6 + number7 + number8 + number9
FROM KMON_SS_INQ
WHERE epoch=1
```

Conversely, a long, narrow table with multiple rows per log record could require many join operations to reconstruct the original log record, which might be a common user request, but aggregates could be expressed naturally using the SQL aggregation operators. Then the query is written as

```
SELECT SUM(number)
FROM KMON_SS_INQ
WHERE qno < 10 and epoch=1
```

We expect that the query optimizer will hide any substantial query performance impact of the different table organizations, leading us to choose the long-and-narrow table layout for tables in which the user may wish to aggregate across statistics collected during a single epoch (for snapshots) or event (for events), *e.g.*, KMON_SS_INQ.

6.2 Monitoring database views

While a SPADE user could issue queries directly to the base tables described Section 6.1, each of those tables stores entries per epoch rather than per application-level request (an HTTP request in the case of Apache). Since we expect SPADE to be used to track down the source of application-level performance variations, it is important for the user to be able to easily view aggregate statistics over the lifetime of an application-level server request. The primary goal of SPADE's views, then is to aggregate data on a per-request basis and to present a set of tables with the same columns as the base tables but storing the statistics as per-request aggregates rather than per epoch. Aggregation is performed by joining a base table (indexed by epoch) with the KMON_APACHE table (which maps each application-level request to the epochs during which the request ran) and aggregating, for each request in the KMON_APACHE table and each column, all entries from the base table with epochs matching those during which the request ran.

Although statistics are themselves easily aggregated across epochs to produce a total number for a particular request (*e.g.*, the total number of TCP packets sent during the handling of a request can be computed by summing the PKTSSENT column of the KMON_NETSTATS table across all epochs during which the request ran), representing per-request aggregates in a meaningful way is challenging because most statistics are collected as absolute values rather than as rates. Since contention for an operating system service or hardware device can be measured in our system as requests for that service or device per time unit, it makes sense to represent most statistics on a per-time-unit basis. For example, a SPADE user might be interested in knowing whether an excessive number of received TCP packets is causing one server request to take longer than an identical request issued at a different time. The base table for

network statistics collects the number of TCP packets received per epoch, but the useful statistic for the user is the total number of packets *per second* received during the lifetime of the request under consideration.

Of course, some statistics count space or items rather than events, *e.g.*, the number of file system buffer queue buffers free at the end of each epoch or the number of processes in the run queue at the end of each epoch. In such cases we use the minimum, maximum, or average over all epochs since a rate metric is not sensible. When counting utilization of such resources (*e.g.*, number of runnable processes) we generally use average or maximum (since maximum contention corresponds to maximum measured utilization) and when counting free resources (*e.g.*, number of free buffer queue buffers) we generally use average or minimum (since maximum contention corresponds to minimal free resources). To provide a unified set of aggregates, SPADE's views provide what we believe to be the appropriate per-request aggregations (generally average event rates over the lifetime of a request). But the system also allows users to directly access the base tables in order to perform their own per-request aggregations and to create their own views and queries.

7 Performance Analysis Using Queries over the System Monitoring Database

The key benefit of using a system monitoring database as a central repository of performance data is that it vastly simplifies the offline process of system performance analysis. Part of this improvement in "ease-of-use" comes from the fact that the database acts as a central repository for performance data collected from all levels of the system (including the application), and from the fact that every datum is keyed by the epoch and time at which it was collected. But the primary reason that the database simplifies performance analysis is that it provides a powerful, flexible query language that lets the user easily cope with the complexity and dimensionality of the massive performance data set. Using simple declarative SQL statements, the user can easily summarize performance data, progressively drill-down to examine the performance of a particular part of the system or application in increasing detail, join together seemingly unrelated data to search for unexpected correlations, or test hypotheses as to the cause of performance problems by searching for time periods during which those problems were exhibited and examining the appropriate system statistics. Most importantly, the user can perform all of these tasks in a unified framework, without having to write ad-hoc tools or scripts to process unwieldy flat data files, and without the constraints of typical performance analysis tools that restrict their data collection and analysis in order to simplify the final presentation of their data.

In this section, we give three examples of how an RDBMS's query language support simplifies typical performance analysis tasks. We begin by illustrating the process a user might follow to investigate application performance anomalies, starting with an overall performance summary and drilling down to look for correlations. We then examine a different kind of analysis that uses different forms of aggregation to search for system-wide performance bottlenecks and to investigate the causes of those bottlenecks. Finally, we present an example of hypothesis-testing, and demonstrate how database queries can be used to confirm a hypothesis and suggest appropriate performance-tuning actions. All of our examples use a database loaded with data from a traced execution of an instrumented version of the Apache 1.3.3 WWW server running atop our instrumented NetBSD 1.3.3 kernel. The server was run on a x86 machine with an AMD K6-2/350 CPU, 128 MB of DRAM (64 of which were dedicated to the trace buffer), and an IBM 9ZX 10,000RPM SCSI disk. The server was driven with a workload of approximately 1000 HTTP 1.0 GET requests issued by a multithreaded client supporting 32 outstanding connections. To minimize variance due to the network, the client was run on the same machine as the server. The SQL shown below follows the Microsoft SQL Server 6.5 syntax.

7.1 Investigating per-request application performance

In this section, we give an example of the process a user of our system might follow in trying to understand the performance differences between different application requests (in this case, HTTP requests). In this example, we are not explicitly looking for a problem with the system or something that needs to be fixed or tuned; instead, we are simply trying to investigate the system's performance in order to get a better understanding of what factors do and do not influence performance.

We begin our investigation by examining the various Apache HTTP requests in order to isolate those which showed the most variation in performance; we will then investigate those requests further to see what was happening inside the system during the servicing of those requests. We start with the following query, which finds requests with a large variance between their minimum and maximum response time. Note that we group requests on their `hash` field; this field is a numeric hash of the `fname` (URL) field used to support databases that cannot group by a `varchar` field. Note also that the `dev` field should really be the standard deviation, but SQL Server does not support the `stddev()` aggregation function in version 6.5):

```
 SELECT a.hash, count(*) AS cnt,
       min(a.resptime_noc) AS minresp,
       max(a.resptime_noc) AS maxresp,
       avg(a.resptime_noc) AS avgresp,
      (max(a.resptime_noc)-min(a.resptime_noc))/avg(a.resptime_noc) AS dev,
      max(a.fname)
FROM kmon_apache a
GROUP BY hash
HAVING count(*) >= 2
ORDER BY dev
```

This query returns a table listing every page accessed more than once during the trace, ordered by the "variance" in response time. The first and the last few rows of this table are shown below:

```
hash        cnt  minresp  maxresp  avgresp  dev     fname
----------- ---- -------- -------- -------- ------- ------------------------------
-1210678261 2    0.0925   0.0929   0.0927   0.0043  /General/Icons/_WL
[...103 rows elided...]
552383629   15   0.0068   0.3649   0.0875   4.0919  /SDG/Software/Mosaic/Docs/_E/_F
-224651581  130  0.0069   0.4548   0.0892   5.0232  /SDG/Software/Mosaic/_B
-125134359  83   0.0076   0.4687   0.0786   5.8698  /SDG/Software/Mosaic/_L
```

The result of this query gives an impression of the behavior of the various requests in the workload. It shows that there are some queries (like the first listed above) with a very low variance in response time, and some with very high variance, like the last few. After examining this summary information, the next step is to try to understand the source of this variance by drilling down and examining the set of requests for one page that demonstrated a high variance.

To do this, we want to extract all of the requests corresponding to one specific hash value. In particular, we want to look at the statistics in other tables that correspond to the time period during which the requests corresponding to one hash value were running. We start this process by building a temporary table holding the request numbers corresponding to one hash, in this case –125134359:

```
SELECT seqno INTO temp2
FROM kmon_apache
WHERE hash = -125134359
```

We can then issue a generic query of the following form to bring all of the appropriate data together:

```
SELECT b.*, a.resptime_noc
FROM table b, kmon_apache a
WHERE b.queryno = a.seqno
  AND b.queryno in (SELECT seqno FROM temp2)
ORDER BY a.resptime_noc
```

where `table` is replaced by the name of one of the materialized views described above, for example `sumstats` or `netstats` or `vmstats`. This type of query makes it easy to visually detect patterns and correlations between response time and other statistics. It also provides an easy way to see exactly what was happening in the system during a particular request. As an example, the following is an excerpt of the above query for the `sumstats` table for two requests, showing an increase in context switches per second and hardware interrupts per second during the second (slower) request relative to the first:

```
queryno     nctxsw/sec     nhwintr/sec     resptime_noc  [...]
----------- -------------- --------------- ------------- ------
[...]
445         714.688        95.667          0.1777        [...]
399         2410.924       142.949         0.4687        [...]
```

Besides attempting to correlate high-variance requests directly with other system statistics, another approach to investigating per-request application performance is to systematically break down each request into its constituent system calls and to study the behavior of those system calls using similar techniques to those we used on a full-request granularity above. We now give an example of this approach using the same example request set as above (those requests with `hash=-125134359`). The first step in this process is to try to isolate the "system call fingerprint" of an HTTP GET request. Obtaining this fingerprint will allow us to isolate a particular request's actions, even if it takes place during a time period in which multiple requests are interleaving system calls. We can obtain this fingerprint by executing the following query, which looks at the set of system calls for one process starting at the beginning of an epoch. This query isolates the system call fingerprint for a request because any server process is only serving one request at a time, and that process initiates an epoch as the first action in serving a request:

```
SELECT *
FROM kmon_syscall s
WHERE s.epoch >= 75 and s.epoch <= 76 -- note: req 38 executed entirely in epoch 75
  AND pid IN (SELECT pid FROM kmon_syscall WHERE time =
                (SELECT min(time) FROM kmon_syscall WHERE epoch = 75)
               AND epoch = 75)
```

The results of the query give the following fingerprint (excerpted from the result table):

```
epoch   name                          arg1        arg2        rval
------- ----------------------------- ----------- ----------- -----------
75      kmon_increment_epoch          61455       -272640744  74
75      read                          3           438404      40
75      sigaction                     30          -272649040  0
75      gettimeofday                  -272649052  0           0
75      __stat13                      596412      590068      0
75      open                          596412      0           4
75      mmap                          0           22719       1074688000
75      writev                        3           -272640784  22969
75      close                         4           0           0
75      gettimeofday                  -272648932  0           0
75      write                         17          597348      92
76      kmon_increment_epoch          61455       -272640416  75
```

```
76      shutdown                        3           1           0
```

Both the `read` and `writev` system calls correspond to network I/O (to read the request and write the response to the socket with file descriptor 3); the `writev` may also cause disk I/O, as it touches the memory-mapped pages of the requested file (file descriptor 4) in copying them to the network socket.

Having identified Apache's HTTP request service fingerprint, we can now go and look at the full system call trace for a given request (or set of requests) in order to examine the performance breakdown of those requests. An example set of queries to do this for four particular requests is:

```
SELECT * FROM syscall WHERE queryno = 237 ORDER BY pid –- response time = 0.0081
SELECT * FROM syscall WHERE queryno = 14  ORDER BY pid –- response time = 0.0522
SELECT * FROM syscall WHERE queryno = 623 ORDER BY pid –- response time = 0.0806
SELECT * FROM syscall WHERE queryno = 445 ORDER BY pid –- response time = 0.1777
```

Note that these four requests were chosen by examining the subset of the `kmon_apache` table corresponding to the hash value described above, and selecting four queries that covered a range of response times. The results of those queries are not reproduced here for space reasons, but they do indicate that the (network) `read()` system call dominates the performance of both request #623 (consuming 0.0709 seconds) and of request #445 (consuming 0.1686 seconds); these are the slowest of the queries. The next, and final, step is to try to determine what is making these calls slow.

This last step illustrates the power of the system monitoring database: whereas traditional performance analysis tools can provide either a system call trace or coarse-grained summary statistics, they cannot correlate a particular subset of a system call trace with summary statistics on the fine-grained timescale of one application request or epoch. However, a query like the following does exactly that (in this case extracting the rate at which data was being sent through the network layer during the epoch in which the slow `read()` call of request #445 was taking place):

```
SELECT b.databytessent/b.time AS datarate
FROM kmon_netstats b
WHERE b.epoch IN
   (SELECT epoch
    FROM kmon_syscall
    WHERE epoch >= (SELECT firstepoch FROM kmon_apache WHERE seqno = 445)
      AND epoch <= (SELECT lastepoch FROM kmon_apache WHERE seqno = 445)
      AND pid = 283
      AND num = 3)
```

This query produces the number 5385901.6, indicating that the network was very busy during the time of that system call. Similar queries with less restrictive `WHERE` clauses can be used to compare this value to the data rates during other epochs, or to the average data rate (1079827.7, obtained via "`SELECT avg(databytessent/time) FROM kmon_netstats`").

7.2 Investigating system-wide performance

In this section we demonstrate the usefulness of SPADE for investigating system-wide performance bottlenecks that impact application performance. We know from previous analysis that Apache invokes the `writev()` system call to send a web page over the network to the requesting client. We will find out whether this system call represents a large fraction of the total time the application spends in the operating system, and if so whether the system call exhibits a large variation in execution time. Assuming both are the case, we will attempt to identify the cause(s) of above-average execution time for this system call by examining the simultaneously-occurring operating system.

First we investigate whether `writev()`contributes significantly to the amount of time Apache spends executing operating system code. The following query computes the total time spent executing each type of system call during the time the Apache server was traced.

```
SELECT name, sum(elapsedtime) as t
FROM kmon_syscall
GROUP BY name
ORDER BY t
```

```
name                    t
----------------------- ------------------------
__fstat13               0.0
break                   0.0
fcntl                   0.0
fstatfs                 0.0
geteuid                 0.0
getpid                  0.0
getsockname             0.0
[...]
bind                    0.162400000000001
open                    0.282499999999989
__stat13                3.21690000000012
accept                  14.5516
kmon_increment_epoch    14.5988000000001
writev                  16.0345
write                   35.1766000000004
nanosleep               58.4613
read                    60.5455000000009
sigsuspend              74.6053
flock                   341.070400000001
select                  510.614999999999
```

We see that `writev()` indeed represents a nontrivial amount of operating system execution time. Next we want to find out whether there is a significant variation in the response time of the `writev()` system call, and if so we want to identify one of the calls that took a significantly longer-than-normal amount of time to complete so that we can investigate it further. The following query lists the elapsed time, last epoch, and first argument to all occurrences of the `writev()` system call that took longer than 0.01 seconds to complete.

```
SELECT s.name, s.arg1, s.elapsedtime, s.epoch
FROM kmon_syscall s
WHERE name = ‘writev’ and elapsedtime > 0.01
ORDER BY elapsedtime
```

```
name                    arg1        elapsedtime             epoch
----------------------- ----------- ----------------------- -----------
writev                  3           0.0101                  762
writev                  3           0.0101                  1412
writev                  3           0.0101                  1885
writev                  3           0.0102                  1159
[...]
writev                  3           0.4325                  1018
writev                  3           0.433                   1959
writev                  3           0.4599                  673
writev                  3           0.6114                  875
```

There is indeed a large variation in execution time, so we decide to investigate one of the two instances of this call with the largest response time, namely

```
writev                  3          0.4599                  673
```

To do this, we want to sum potentially-relevant statistics over all epochs during which the system call executed. We will examine the `kmon_sumstats` table for possible explanations of the long execution time of the call. We know that the call finished at epoch 673 and started 0.4599 seconds earlier, so we first need to find the epoch during which the call began. This information could have been stored explicitly in the `kmon_syscall` table, but we can recalculate it fairly easy by summing the time field of any summary snapshot table (since one snapshot is taken per epoch, there will be an epoch duration entry for each epoch in any such table) starting at various epochs and ending at the ending epoch of the system call under consideration, until we find the starting epoch that causes the sum of epoch durations to equal the duration of the system call. Using this method we find that the system call begins began in epoch 598.

Now that we have identified the relevant span of epochs, we will examine the `sumstats` table to find a possible explanation for the long running times of this system call.

```
SELECT x.*
FROM kmon_sumstats x
WHERE x.epoch >= 598 and x.epoch <= 673
```

Since the full output does not fit easily on a printed page, we extract the interesting columns and list only them below:

```
epoch nctxsw   ntraps  nsyscalls  nhwintrs  nswintrs  nfaults  pzfod  nzfod_created
----- -------- ------- ---------- --------- --------- -------- ------ -------------
598   6        3       21         0         3         0        0      0
599   6        11      19         0         3         8        0      0
600   2        2       13         0         1         1        0      0
601   2        2       15         1         1         1        0      0
602   2        2       15         0         1         1        0      0
603   4        4       13         1         3         1        0      1
604   2        3       13         0         1         2        0      0
605   5        11      23         0         2         9        0      0
606   2        2       14         0         1         1        0      0
607   2        2       14         0         1         1        0      0
608   2        7       13         1         1         6        0      0
609   2        4       14         0         1         3        0      0
610   2        2       14         0         1         1        0      0
611   2        7       13         0         1         6        0      0
612   2        7       13         0         1         12       6      0
613   2        2       14         0         1         2        1      0
614   4        4       18         1         3         2        1      0
615   1        1       11         1         0         2        1      1
616   8        4       26         0         4         0        0      0
617   6        3       18         0         3         0        0      0
618   6        3       18         0         3         0        0      0
619   6        3       18         1         3         0        0      0
620   6        3       19         0         3         0        0      0
621   6        3       18         0         3         0        0      0
622   6        3       18         0         3         0        0      0
623   6        3       18         0         3         0        0      0
624   6        3       18         1         3         0        0      0
625   5        4       21         0         3         1        0      0
626   1        0       6          0         0         0        0      0
627   2        3       12         0         2         1        0      0
628   1        0       6          2         0         0        0      0
```

629	0	6	11	0	0	9	3	0
630	3	2	15	0	1	1	0	0
631	4	3	15	2	2	2	1	1
632	4	4	14	1	2	2	1	1
633	2	4	13	0	1	6	3	0
634	16	11	67	1	8	3	0	3
635	8	6	22	0	4	2	0	11
636	1	1	6	0	1	0	0	0
637	1	0	8	2	0	0	0	0
638	1	1	6	0	1	0	0	0
639	1	1	6	0	1	0	0	0
640	5	13	17	3	1	12	0	0
641	10	4	30	0	4	0	0	0
642	8	6	20	0	4	2	0	0
643	6	3	18	1	3	0	0	0
644	6	3	18	0	3	0	0	0
645	8	4	25	0	4	0	0	0
646	6	3	18	0	3	0	0	0
647	6	3	19	0	3	0	0	0
648	6	3	20	1	3	0	0	0
649	36	2669	733	9	5	2873	172	58470
650	43	484	143	1	3	498	16	7770
651	39	27	23	1	4	23	11	3
652	38	33	22	1	3	30	12	3
653	39	36	19	1	4	32	12	3
654	3	9	16	1	1	8	1	1
655	2	2	15	1	1	1	0	0
656	1	1	6	1	1	0	0	0
657	4	4	15	2	2	2	0	2
658	2	2	14	0	1	1	0	0
659	8	13	32	2	4	10	1	3
660	2	2	7	1	2	0	0	0
661	1	1	6	0	1	0	0	0
662	2	2	14	0	1	1	0	0
663	2	0	9	0	0	0	0	0
664	3	54	19	0	1	54	2	0
665	2	54	17	0	1	53	2	1
666	2	59	16	0	1	58	2	0
667	40	28	34	0	4	24	11	3
668	38	33	23	0	3	30	12	3
669	38	33	23	1	3	30	12	3
670	38	33	22	1	3	30	12	3
671	39	34	23	1	4	30	12	3
672	37	63	30	2	3	60	3	0
673	4	29	9	0	1	28	11	3

We see a large spike in all of these statistics near the end of execution of the `writev()` call. The increased number of context switches (`nctxsw`) and on-demand zero-filled pages (`pzfod` and `nzfod_created`) around epoch 649 suggests that many new processes may have been created during those epochs, which could have led to the increased response time for the `writev()` call. To find out if this is the case, we issue a query to find any calls to `fork()`made during these epochs:

```
SELECT x.epoch, x.name, x.elapsedtime
FROM kmon_syscall x
WHERE x.epoch >= 598 and x.epoch <= 673 and x.name = ‘fork’
```

This query shows that there were 32 calls to the `fork()` system call while the `writev()` call was executing, all of which occurred during epoch 649. In comparison, there were only 63 calls to `fork()` during the entire trace (as determined by a query `like SELECT count(*) FROM kmon_syscall WHERE name=’fork’`).”

The increased execution latency of `writev()` contributed by these 32 calls to `fork()` was not due primarily to time spent executing the `fork()` call itself, though, since time spent executing `fork()` only accounts for

```
SELECT sum(x.elapsedtime)
FROM kmon_syscall x
WHERE x.epoch >= 598 and x.epoch <= 673 and x.name = 'fork'

0.0109
```

seconds. Instead, the slow `writev()` response time is due to the many new processes in the system resulting from the `fork()`, which cause an increase in context switches, page faults, and on-demand page zero-filling. This analysis leads us to conclude that the performance of the `writev()` system call could, in at least some cases, be improved by a more efficient implementation of the system activities associated with `fork`ing. From an application perspective, Apache could be less affected by the sensitivity of `writev()` performance to forking by using threads instead of processes (which would reduce the amount of forking going on in the system while Apache executes). It is conceivable that some of the page faults we see during and after epoch 649 are caused by the `writev()` call itself (since writing to the network from a memory region mapped from a file, when a portion of the file to be written to the network has not already been read into the virtual memory cache, will cause page faults as the file is read into the virtual memory cache). However, the strong temporal correlation with `fork()` calls suggests that the `fork()` calls are the primary cause of the page faults, not the application faulting in web pages by calling `writev()`. Finally, we note that we can attribute this forking activity to Apache (as opposed to other system processes) by examining the `pid` of the process calling `fork()` in epoch 649; it is indeed the `pid` of one of the Apache processes.

7.3 Hypothesis testing

For our last example, we consider the typical performance analysis task of hypothesis testing. The goal here is to formulate a hypothesis that potentially explains some performance anomaly, and then to use the collected performance data to verify or refute that hypothesis. In this example, we will start with an already-formed hypothesis: that disk writes (due to Apache adding entries to its log file) interfere with the service time of HTTP requests. We will demonstrate how simple queries allow the user to confirm this hypothesis and determine the root cause of the interference, and we will show how this knowledge permits the user to develop a few simple system tweaks that might improve overall performance.

The first step in the hypothesis-testing process is to construct an aggregation query to test whether the hypothesis is supported at the coarsest level. In this case, we construct a query that calculates the average response time for HTTP requests during which a write I/O occurred and compares that time to the average for HTTP requests during which only read I/Os occurred. The query is presented below; note that `bioevents.rw` is `1` during requests that contain at least one write I/O:

```
SELECT avg(a.resptime_noc) AS time, b.rw
FROM kmon_apache a, bioevents b
WHERE a.seqno = b.queryno
GROUP BY b.rw
```

This query returns the following output:

```
time             rw
------------     --
0.14513          0
0.59088          1
```

Requests during which writes occurred take on average more than four times longer than queries with no writes! Clearly, the presence of writes is negatively affecting HTTP service time (note that the writes are not part of the requests themselves, since all requests are GETs).

Given this initial confirmation of our hypothesis, the next step is to formulate a query that drills down and displays information about each HTTP request that was running at the time when a disk write occurred. In addition to selecting the request's URL and URL hash, we use the database's data manipulation facilities to group requests by their URL hash, to compute the number of requests containing writes, the min, max, and average response times for the queries containing writes, and the overall average response time. These statistics allow us to examine the performance difference between requests for the same page that did or did not contain a write. The query is:

```
-- build temporary table with counts, response time
SELECT hash, count(*) AS cnt, avg(resptime_noc) AS avgresp
INTO temp_cnt1
FROM kmon_apache
GROUP BY hash
GO

-- perform query
SELECT a.hash,
       count(*) AS num_w_write,
       max(t.cnt) – count(*) AS num_wo_write,
       avg(t.avgresp) AS overall_avgresp,
       avg(a.resptime_noc) AS avgresp,
       min(a.resptime_noc) AS minresp,
       max(a.resptime_noc) AS maxresp,
       max(a.fname) as fname
FROM kmon_apache a, temp_cnt1 t
WHERE seqno in (SELECT queryno FROM bioevents WHERE rw = 1)
  AND t.hash = a.hash
GROUP BY a.hash
ORDER BY avgresp
GO
```

This query produced output of the form below (only three of the 33 returned rows are shown):

```
hash        num_w_write num_wo_write overall_avgresp avgresp minresp maxresp fname
----------- ----------- ------------ --------------- ------- ------- ------- ------
-224651581  2           128          0.0892          0.2079  0.0679  0.3479  /file1
770285138   1           4            0.0999          0.2924  0.2924  0.2924  /file2
-1851629361 1           6            0.1016          0.2959  0.2959  0.2959  /file3
```

This data shows that there is a general trend across requests that requests with writes take longer than requests without writes. The fact that the overall average response times are low (roughly on par with the global average of 0.14513 for queries without writes computed above) while the response times with writes are high confirms that the large service times for requests with writes are a property of the writes and not of any peculiarities of the pages requested while writes were occurring.

At this point in the analysis, the hypothesis is essentially confirmed: writes do impact the response time of requests active while the writes are occurring (even though the writes are not generated by those requests). The next step is to understand precisely why the writes have such an impact on performance. To do this, we drill down further by selecting one page (that with hash 770285138 in this case) to examine in detail. We start with a set of queries that select general information about each request for that page (from the `kmon_apache` table) and some of the interesting I/O aggregate statistics:

```
SELECT * FROM kmon_apache WHERE hash=770285138 ORDER BY resptime_noc

SELECT seqno
INTO temp3
FROM kmon_apache
WHERE hash = 770285138
GO

SELECT b.*, a.resptime_noc
FROM bioevents b, kmon_apache a
WHERE b.queryno = a.seqno
  AND b.queryno IN (SELECT * FROM temp3)
ORDER BY a.resptime_noc
```

These queries produce the following output:

```
seqno    resptime_noc  firstepoch  lastepoch  hash        fname
-------- ------------- ----------- ---------- ----------- -------------
107      0.0214        214         216        770285138   /demoweb/_CB
510      0.0271        1022        1024       770285138   /demoweb/_CB
944      0.0574        1891        1896       770285138   /demoweb/_CB
770      0.1013        1540        1550       770285138   /demoweb/_CB
413      0.2924        822         842        770285138   /demoweb/_CB

queryno     size      servicetime  rw  resptime_noc
----------- --------- ------------ --- -------------
107         7168.0    0.0091       0   0.0214
944         8192.0    0.025        0   0.0574
413         6332.63   0.3844       1   0.2924
```

As can be seen from the second output table, request #413 (the only request containing a write) has a much higher I/O service time than the other two requests containing just reads. Note that the `servicetime` field includes the total I/O time for all requests that *completed* during the request, and as such can be larger than the request's response time.

The last step in the process is to drop one further level of aggregation and look at the individual I/O requests made during each of the three requests in order to determine why the I/O service time is so much higher for the request with a write. The following queries dump the I/O traces from each of the three requests:

```
SELECT b.epoch, b.rw, b.size, b.pblkno, b.pblknodiff, b.servicetime, a.resptime_noc
FROM kmon_bioevents b, kmon_apache a
WHERE b.epoch >= a.firstepoch AND b.epoch <= a.lastepoch AND a.seqno = 107

SELECT b.epoch, b.rw, b.size, b.pblkno, b.pblknodiff, b.servicetime, a.resptime_noc
FROM kmon_bioevents b, kmon_apache a
WHERE b.epoch >= a.firstepoch AND b.epoch <= a.lastepoch AND a.seqno = 944

SELECT b.epoch, b.rw, b.size, b.pblkno, b.pblknodiff, b.servicetime, a.resptime_noc
FROM kmon_bioevents b, kmon_apache a
WHERE b.epoch >= a.firstepoch AND b.epoch <= a.lastepoch AND a.seqno = 413
```

These queries produce the following output:

```
epoch  time    rw  size   pblkno     pblknodiff  servicetime  resptime_noc
-----  ----    --  ----   ------     ----------  -----------  ------------
215    0.0033  0   7168   3562016    3288656     0.0091       0.0214

epoch  time    rw  size   pblkno     pblknodiff  servicetime  resptime_noc
-----  ----    --  ----   ------     ----------  -----------  ------------
1896   0.0001  0   8192   3225760    82816       0.0082       0.0574
1896   0.0094  0   8192   2166016    1059744     0.0168       0.0574

epoch  time    rw  size   pblkno     pblknodiff  servicetime  resptime_noc
-----  ----    --  ----   ------     ----------  -----------  ------------
822    0.0035  1   8192   155296     32          0.0123       0.2924
822    0.0075  1   8192   194176     38880       0.0108       0.2924
822    0.0173  1   8192   160        194016      0.0137       0.2924
824    0.0006  1   8192   271760     271600      0.0209       0.2924
824    0.0071  1   8192   271776     16          0.0176       0.2924
825    0.0033  1   8192   271792     16          0.0129       0.2924
825    0.0047  0   1024   274014     2222        0.0079       0.2924
825    0.0061  0   3072   274144     130         0.0027       0.2924
825    0.0062  0   2048   274150     6           0.0015       0.2924
825    0.0064  0   2048   274154     4           0.0003       0.2924
825    0.0065  0   3072   274160     6           0.0003       0.2924
825    0.0067  0   4096   274166     6           0.0003       0.2924
825    0.007   0   6144   274176     10          0.0004       0.2924
826    0.0029  1   8192   543376     269200      0.0065       0.2924
827    0.0007  1   8192   811632     268256      0.0172       0.2924
827    0.0082  1   8192   928048     116416      0.0185       0.2924
828    0.0016  1   8192   966848     38800       0.0131       0.2924
828    0.0088  1   8192   1160832    193984      0.0128       0.2924
828    0.0177  1   8192   1351520    190688      0.016        0.2924
828    0.0281  1   8192   1506608    155088      0.0193       0.2924
828    0.0326  1   8192   1545520    38912       0.0149       0.2924
828    0.0409  1   8192   1623008    77488       0.0128       0.2924
828    0.05    1   8192   1700720    77712       0.0173       0.2924
828    0.0601  1   4096   1852464    151744      0.0192       0.2924
828    0.0667  1   8192   1852480    16          0.0168       0.2924
828    0.0723  1   1024   1853662    1182        0.0122       0.2924
828    0.0786  1   8192   2162976    309314      0.0119       0.2924
828    0.085   1   8192   2162992    16          0.0127       0.2924
828    0.0932  1   4096   2625136    462144      0.0146       0.2924
828    0.098   1   8192   2625280    144         0.013        0.2924
828    0.1047  1   8192   2632384    7104        0.0116       0.2924
829    0.0006  0   8192   4286848    1654464     0.0135       0.2924
829    0.0007  0   4096   274192     4012656     0.0069       0.2924
829    0.0009  0   5120   274208     16          0.0003       0.2924
829    0.0011  0   3072   274224     16          0.0004       0.2924
829    0.0012  0   4096   274240     16          0.0003       0.2924
829    0.0015  0   8192   274256     16          0.0005       0.2924
829    0.0018  0   5120   274288     32          0.0005       0.2924
```

We observe a few interesting patterns. First, the one I/O in epoch 215 for the first request is most likely the original read of the requested page (the page's file size is 6169 bytes, which, when rounded up to a multiple of the 1KB fragment size used in our file system, results in an I/O size of 7168). Since the data set was small enough to be mostly cached, and since that physical block number never reappears, we can assume that all other I/Os in the output are not generated by the requests in question. The most interesting patterns shows up in the third output table (for request #413, the one with the write). First, notice that the write I/Os tend to have significantly larger service times than the read I/Os. Notice also that the writes are, for the most part, non-sequential, incurring large seeks (the value of `pblknodiff` indicates the

approximate seek distance in blocks). The large seeks combined with the fact that writes are not completed until they have been physically written to the disk medium probably account for the long write times. The writes also interrupt a string of very tightly-clustered reads; in this case, the reads are delayed by the time for the intervening writes to complete, although the performance of any individual read is not significantly degraded (probably because our high-end disk has a very large track buffer that services all but the first few of those reads), but with less sophisticated disks the interrupted reads would most likely suffer in performance as well.

One final important pattern to notice is that when the writes occur, a large number of them occur in direct succession. Further analysis of the data stored in the `kmon_bioevents` table (using a query such as `"SELECT DISTINCT epoch FROM kmon_bioevents WHERE rw = 1"`, results omitted here for space reasons) shows that all of the writes in the trace occur within a 25-epoch span covering 0.32 seconds of real time. This suggests that we are seeing the effects of the NetBSD file system buffering policy. In NetBSD, as is typical in a UNIX-like system, writes are absorbed by the buffer cache and sent to disk in a single batch whenever the system `update` daemon periodically wakes up.

At this point we have thoroughly verified our initial hypothesis and have investigated its cause. We have determined that the poor response time of queries during which writes took place is due to a combination of the overhead of large seeks, potentially unbuffered writes at the disk, and the fact that a large number of writes are sent to disk at once, blocking the progress of other queries running during that time. One task remains: to figure out what we might do to lessen the performance impact of writes. One simple solution would be to direct writes to a separate I/O subsystem. In the case of Apache, the only writes being generated are Apache's own log updates, so by moving the log files to a separate disk on a separate controller, the writes would not be able to interfere with the I/O of other running queries. If adding hardware is not an option, another solution would be to modify the OS buffer cache write policy. Since response time is the key metric for applications like web servers, it would be preferable to spread the writes out in time rather than batching them together. If this were done, more requests would show degraded performance, but the degradation would most likely be much smaller, and potentially would be lost in the noise of variations in network latency. This analysis suggests that the standard OS buffer cache policy is wrong for a single-application server running a web server, and that the cache policy for such a system should be tuned to drain writes to disk shortly after receiving them, rather than waiting for the `update` daemon to run.

Thus we have shown in this example that our system monitoring database provides a powerful tool for hypothesis testing and investigation. We have also seen an example of how the system makes it possible to detect performance problems and to identify possible approaches for tuning. Note, however, that we began with an already-formulated hypothesis. A natural extension would be to use data mining techniques to generate new hypotheses from the trace data set, and to then use the process just described to test and act on those hypotheses.

8 Suitability to Task

In the course of developing SPADE we investigated the suitability of four databases for supporting monitoring-data storage and querying: PostgreSQL 6.4.2, Microsoft Access 97, IBM DB2 5.2 for Linux, and Microsoft SQL Server 6.5. Our dataset poses a reasonably challenging workload for these databases—for example, our longest table had more than 175,000 rows and our widest table had 120 columns.

Our primary criteria in selecting a database were, in rough order of decreasing importance,

- robustness (*i.e.*, no serious bugs tickled by our workload)
- sufficient speed for bulk loading, view creation, and querying
- user interface support for browsing large tables
- support for the necessary datatypes

PostgreSQL is an open-source object-relational database based on the POSTGRES database developed at the University of California, Berkeley. We found its bulk loading performance to be acceptable, but it crashed when processing queries to generate our "views." Another problem with this database was its text-mode front end, which limits easily-viewable table widths to the largest window width displayable on the user's terminal (*i.e.*, there is no ability to scroll horizontally). Additionally, PostgreSQL did not allow primary keys on 64-bit values, which we wanted to use for our timestamps (since the log records use the 64-bit CPU cycle counter as their timestamp). This forced us to split the timestamp into two unsigned 32-bit quantities. Finally, PostgreSQL provided very poor error reporting during bulk loading. In particular, any error or inconsistency in an input table caused the connection from the PostgreSQL server to the client to be unceremoniously dropped, resulting in a cryptic error message from the client and no indication of which line in the input file caused the error. In one case the "error" was not really an error in our input file at all, but rather a bug in PostgreSQL. On the other hand, PostgreSQL does have at least one nice feature for our application, namely the `SELECT INTO` command that allows a new table to be created directly from a base table without requiring explicit declaration of a schema for the new table. Types and column names are inferred from the `SELECT` statement that generates the new table. But due to its inability to handle our queries, we were unable to use PostgreSQL.

Microsoft Access 97 is the relational database component of the Microsoft Office suite. We found its bulk loading to be relatively fast, but the impossibility of scripting the bulk loading process made repeated bulk loading (necessary as we gathered and examined various traces from the web server) painful. The most serious drawback of Access was its inordinately slow query speed—generating a view table took several minutes in some cases. On the other hand, Access supports an easy-to-use graphical user interface and allows horizontal scrolling via its spreadsheet-like display mode.

DB2 is IBM's object-relational database product. We found its bulk loading to be slow and its query response time to be very slow. This latter fact surprised us, but we suspect that DB2 would have exhibited much faster response time had we tuned the appropriate configuration parameters. A relatively thorough search of the documentation did not reveal the appropriate parameters to tune, so we gave up on DB2. Another problem we had with DB2 that could probably have been solved by issuing the correct administrative command was DB2's running out of transaction log space during creation of materialized views. We suspect that DB2 was logging the changes made by each table insert operation, requiring a large amount of log space for each `SELECT INTO` query. Unfortunately we were unable to determine the proper mechanism for increasing the available transaction log space or telling DB2 to disable transactions for `SELECT INTO` operations. Another drawback of DB2 is its text-mode output which, like PostgreSQL's, prevents horizontal scrolling of wide output tables.

Microsoft SQL Server 6.5 is the object-relational database component of the Microsoft BackOffice suite. We found both its bulk load and query response times to be good. Additionally, SQL Server supports horizontal scrolling of query results, allowing wide tables to viewed relatively easy. We were also quite impressed with SQL Server's support for multiple simultaneous networked clients and its GUI administrative and tuning tools. On the other hand, the product has two serious drawbacks for our application. First, it does not support the standard deviation or median SQL aggregate functions; these would have been extremely useful for finding table columns with high variance and for finding a representative median value for a collection of data points. Second, SQL Server does not allow storage of

64-bit integers, or even of unsigned 32-bit integers. This forced us to represent time as a floating point value rather than as the 64-bit hardware cycle counter value stored with each log record. Moreover, we were forced to use as keys the two 32-bit values we originally introduced to address PostgreSQL's inability to handle 64-bit primary keys, because it is not possible to guarantee that two different 64-bit timestamps will map to unique floating point timestamps. Despite these drawbacks, SQL Server was by far the best database of the four we considered due to its lack of serious bugs, its support for horizontal scrolling, and its excellent out-of-the-box (untuned) performance. In the following table we present the performance of SQL Server on the most common database operations used by SPADE.

Operation	**Time** (min:sec)
run *datapump*	0:49
run *db-insert*	2:08
bulk-load database	3:27 (2000 rows/sec)
create view tables	6:36

The following table summarizes our impressions of the various databases that we examined:

Database	**Robustness**	**Bulk Load Performance**	**View Table Creation Performance**	**User Interface**	**Data Type Support**
PostgreSQL	★	★★★	N/A	★★	★★★
Access 97	★★★	★★★	★	★★★	★★★
DB2	★★	★★	★★	★	★★★
SQL Server	★★★★	★★★★	★★★★	★★★	★★

9 Future Work

SPADE might serve as the basis for future work both within the framework of the current system and in the context of longer-term projects. First, several minor tweaks on the current prototype are possible. Automatic compression of the in-kernel trace buffer would allow *kmond* to run less frequently, imposing lower overhead for periodically writing the buffer to disk. Many of the values collected in the trace are small diffs or equal to zero, suggesting that the compression techniques would work well.[2] Also, *datapump* and *db-insert* could be integrated; we wrote them as separate programs simply because C (used to write *datapump*) is better at manipulating raw binary files (the format used in the log files written out by *kmond*) directly, while Perl (used to write *db-insert*) is better at handling text and file operations.

[2] Of course the overhead of performing this compression must be balanced with benefit of reduced I/O for writing the buffer to disk, but since SPADE is targeted to data-intensive servers that are likely to be I/O bound, and CPU speeds are increasing much faster than is I/O system performance, the tradeoff is likely to favor compression.

Finally, appropriate user-defined (*i.e.*, not standard in SQL) statistical aggregates over some of the table columns might be used to address some of the difficulties we experienced in expressing using SQL meaningful aggregates for some statistics. For example, while it is very useful to know how much seeking a disk performed during the service of a web request, the average seek distance per time (which is the most meaningful summary statistic we could express in SQL for this column) does not tell the whole story—it makes a continuous stream of medium-sized seeks look the same as a small number of blocks of giant-seek-followed-by-sequential-read.

Within the framework of the current system, several follow-on projects to improve online data collection and offline analysis are possible. Fine-grained resource accounting with some notion of application-level accounting units (*e.g.*, individual HTTP requests), such as resource containers [BDM99], would allow resource usage to be attributed precisely to the appropriate application-level entity, allowing more detailed per-request performance analysis. Measuring the time each application-level accounting unit spends waiting for each system resource would provide similarly useful data, and would help greatly in pinpointing the location and severity of system bottlenecks and in assessing the impact of those bottlenecks on response time. Instead of guessing that an HTTP response experienced increased latency because certain system resources happen to be highly contended while the request was being processed, we could know exactly for which resources the request was waiting, and for how long.

In lieu of such accounting, some mechanism for low-overhead continuous fine-grained event recording would at least allow all relevant system events to be logged individually rather than being aggregated into snapshots taken only once per epoch. Unfortunately our monitoring trace buffer already fills up at an alarmingly fast rate, so finer-grained statistics collection would require application of data reduction techniques. Sampling at the granularity of logical application-level units, as suggested in Section 4.3, represents one possible approach to data reduction. Additionally, some statistics that we currently collect may not be useful to any potential subsequent analysis, and some others might be derivable from a small set of base statistics. Identifying the minimal set of statistics that must be collected in order to avoid constraining subsequent analysis offers fertile ground for future investigation. Finally, one might use SPADE as the basis for integrating existing performance analysis tools into a single data collection framework, allowing a range of analyses not possible using any single existing tool.

Any number of additions to our existing offline data analysis framework could potentially allow for even more sophisticated insights than those presented in this paper. For example, we are interested in adapting existing data mining algorithms to find the cause of performance variations and bottlenecks using the data collected by our monitoring tool. Association rule algorithms like Apriori [AS94] work well for binary data (*e.g.*, discovering that when one item is purchased in a transaction, another item is very often also purchased in that transaction) but require substantial modification to work with continuous numeric data of the type collected in SPADE. Existing binary association rules algorithms could potentially be applied to our data set if we could set arbitrary delineations between “normal/acceptable” and “abnormal/unacceptable” performance for each metric and then treat an “abnormal/unacceptable” measurement made during a particular epoch as an item “purchased” in the transaction corresponding to that epoch. But setting these cutoff points seems highly workload dependent and those points are likely to change over time even within a single workload. As another potential addition to our offline data analysis framework, integrating online query processing (*e.g.*, the tools developed in the CONTROL project at Berkeley) into the database query process would alleviate some of the annoyance due to the poor database performance we experienced. It would allow for more sophisticated (*i.e.*, time-consuming) queries to be constructed in the process of exploring possible correlations in the data, since the user would not need to wait for the entire dataset to be processed before initial results would be returned. Indeed the scenarios presented in [HAR99] have strong analogies to the process of probing the SPADE database for performance insight.

A somewhat simpler starting point for improved offline data analysis would be to develop or use existing visualization tools, such as graphing or OLAP packages, for analyzing the SPADE database. Simply graphing two or more statistics with respect to time on the same set of axes could enable visual identification of correlations that are obscured when statistics are represented in purely numeric, tabular form. Though we did not have time to investigate it, Microsoft Excel allows the construction and issuing of SQL queries to a remote database, as well as the importing of the result tables. This facility could be used to load result tables for graphing by Excel's built-in graphing tool.

SPADE might also serve as the first step in a number of longer-term projects that require substantial additions to, or reworking of, the current framework. The insertion of monitoring data into the database could be moved online if the overhead of the loading process could be reduced. While we have shown in this project that existing full-featured databases do not offer adequate performance for continuous realtime insertion of monitoring statistics of the type we collect in SPADE, we are interested in determining whether embedded database backends such as Berkeley DB [Sleepycat] might offer an adequately-performing alternative to full RDBMSs while still offering useful query features like access methods and application-aware caching.

We are also interested in applying the techniques developed in SPADE to guide automatic adaptation by applications and operating systems. The extra system support needed to fully realize this goal includes online data analysis (in addition to the existing online data collection) and triggers. Our goal is develop a system that, guided by a user-specified policy, sets appropriate triggers that fire to invoke adaptation code when the system reaches a state in which adaptation is required. This goal can be thought of as encoding the logic a human administrator would use to manage a system (with respect to both performance tuning and maintenance of availability) using triggers over the database of monitoring data, and appropriate adaptation code to execute in response to those triggers. To this end, we also need to determine what statistics should be monitored to enable automatic system adaptation. We are currently investigating these and other related issues within the context of the ISTORE project [BOK+99].

Finally, the SPADE framework could be extended to collect data from a cluster of nodes, for example to analyze the performance of a cluster-based server such as an Internet search engine. Some mechanism for synthesizing a global time across nodes would be essential to correlating statistics collected simultaneously on different machines, and node identifiers would need to be added to the database schema. Data could be stored locally on each node and bulk loaded into a distributed database running on each node in the cluster, or it could be transferred to a single analysis node (in bulk fashion or continuously as statistics are collected) running a single-node database.

10 Related Work

SPADE is by no means the first system aimed at monitoring system performance and correlating application-level performance variations to their underlying causes. But by using a database as its structuring principle, it does offer the possibility of integrating a number of existing tools and simplifying the construction of new tools.

SPADE might be used as a building block in a system such as that proposed in [SS97]. In that paper the authors suggest continuously profiling an extensible operating system in order to determine when and how the kernel should be adapted via extension. Performance data is deposited into a monitoring database and adaptation is driven by querying that database. Since the goal of this adaptation is to improve application performance, a system like SPADE that collects and correlates application-level behavior with operating system performance data is well-suited to guiding the desired adaptation.

The CARD project [AP97] used relational database technology to monitor computer clusters. Like SPADE, CARD recorded data about CPU, network, and I/O system utilization in a relational database, and allowed users to analyze the data using SQL queries. But unlike SPADE, CARD gathered data at a coarse time scale and did so simultaneously from multiple machines in a cluster, rather than on a fine granularity and from a single node. Additionally, CARD only used data available from user-level administrative utilities; it did not instrument the kernel to obtain additional statistics and traces of operating system events. This difference in approach stems from CARD's goals of enabling system administrators to monitor resource utilization across all the machines they administer and to detect abnormal conditions, in contrast to SPADE's focus on correlating application performance with low-level system behavior to enable a programmer to tune an application and/or operating system.

The use of data mining to *detect* unusual patterns is closely related to SPADE's proposed use of data mining to analyze application performance. For example, in [LSM99] the authors build a system that mines system audit data (*e.g.*, shell commands issued, system calls made, and network connections established or accepted) associated with individual users, in order to distinguish normal system usage from that associated with an intrusion. The goal of this mining differs from that which we propose for SPADE in that [LSM99] seeks to detect abnormal events and assumes an administrator will trace the cause, while the mining we propose for SPADE should both detect abnormal events and provide an indication of their underlying cause. But the two ideas are similar in that they attempt to use data mining to classify monitoring data as indicative of "normal" or "abnormal" system behavior.

Hardware and software performance monitoring tools date back to the earliest computers. A number of standard Unix utilities gather statistics similar to SPADE's "snapshot" data, though their output is designed to be human readable rather than processed automatically, and they provide no facility for correlating the collected data with user-level application behavior or with the data collected by one another. These utilities include *netstat* (network statistics), *iostat* (I/O system statistics), *nfsstat* (network file system statistics), *ps* (per-process accounting statistics), *pstat* (various OS-level data structure statistics), *vmstat* (process, virtual memory, disk, trap, and CPU activity statistics), and *systat* (which combines the functionality of many of the other utilities). Additionally, a number of tools allow applications to be instrumented to collect user-level runtime statistics which are then post-processed to reveal performance bottlenecks. Such tools include gprof [GKM82], QPT [LAR93], Pixie [MIPS90], Atom [SE94], and EEL [LS95]. A third set of tools use hardware performance counters to collect statistics and correlate those statistics to application-level causes. Such tools include VTune [VTUNE] and DCPI [ABD+97]. Once appropriate schema are defined, SPADE allows any tool to be used to gather performance information. Thus any of these existing systems could be used as additional data sources in the SPADE system, increasing the scope and/or detail of the queries that could be made to the database.

Like our web server study, previous projects have attempted to study the causes of application latency. [EWC+96] propose techniques for evaluating interactive system performance by measuring event-handling latency. As we did in our web server study, they focus on latency rather than throughput as key to the user experience. They examined three variants of Windows, replacing the operating system idle loop with their own loop that reads the Pentium hardware counters. Their system also derives information by intercepting events going to and from the Win32 message queue. They did not instrument the operating system, so they were somewhat limited in their ability to correlate application events (as seen through the message queue) to underlying operating system causes.

Other studies have attempted to correlate application performance with the performance of operating system and hardware primitives [BS97, Bro97]; unlike SPADE, however, the systems described in those studies take a static approach, relying on the user to manually isolate the OS and hardware primitives that govern application performance, then predicting the application's performance based on microbenchmark measurements of those primitives. In contrast, SPADE dynamically determines which

OS subsystems and hardware components govern application performance at every point in time, and can thus detect transient bottlenecks due to variations in the patterns of system utilization.

Finally, operating system adaptation based on runtime monitoring has been suggested by a number of researchers. In [PAB+95] the authors describe "optimistic incremental specialization," a policy by which operating system code is incrementally optimized for common cases and dynamically "fixed up" when the assumed constraints are violated. More recently, [SSS96] proposes to profile the behavior of operating system modules in the context of each application that uses them, and to then use this information to re-optimize modules (creating a new copy for each client application) for improved performance when used by that application. Both of these ideas are similar in spirit to the automatic adaptation we propose as future work in SPADE, though they focus on compiler-driven automatic modification to operating system components rather than on explicitly built-in adaptive behavior of applications or operating systems.

While SPADE differs in various ways from the above systems, it provides a framework that could increase the ease of writing new performance monitoring tools and/or the usefulness of many of these existing tools. SPADE provides a standard interface between tools that gather monitoring data and tools that use the data—its schema defines the available data and the data's format, while SQL defines the syntax used for querying and updating the data repository. Thus while our project examined the usefulness of making static queries to a database of performance information collected through relatively simple operating system and application instrumentation, the database framework we have developed offers the potential to enable much more sophisticated monitoring and adaptation using new or existing tools, and to allow these tools to interact and to be composed in ways that are impossible when each tool uses its own data collection, storage, and analysis mechanisms.

11 Conclusions

The tasks of analyzing, understanding, and tuning the performance of large server systems are becoming increasingly important as computing transitions to a model of data-centric infrastructure services backed by large servers. Unfortunately, traditional performance measurement and analysis tools are incapable of meeting the demands of this environment, as they are not designed to provide the detailed, fine-grained, application-correlated data needed to understand and optimize the key user-driven metrics of latency and response time. Furthermore, most of these traditional tools are constrained by an artificial integration of system instrumentation, data storage, and data analysis that limits their flexibility.

In this paper, we presented the architecture and implementation of SPADE, a performance analysis system that overcomes these failings of traditional performance tools by using the facilities of a relational database to mediate the interactions of the software components responsible for instrumentation, storage, and analysis. SPADE's database decouples data collection from data analysis, allowing for flexible analysis unconstrained by the design of the system instrumentation. At the same time, it also provides a centralized store where detailed operating system and application-level performance monitoring data can be integrated, enabling system performance analysis to encompass all levels of the system simultaneously.

To implement the system we instrumented the NetBSD single-node operating system kernel and the Apache web server to collect performance data, wrote a background daemon process and offline utilities to insert the data into a relational database, and created "materialized views" of the resulting tables to correlate low-level operating system behavior with per-request application response time. We showed how a user can write queries over the database to identify application-level server requests and operating system functions exhibiting poor performance and to track down the cause(s) of that poor performance. In particular, we described three such user sessions. First, we showed how a user can

identify server application requests that perform identical operations but exhibit a variation in response time, and can extract insight about the potential underlying causes of that variation by exploring data collected at different levels of the system. We next showed that SPADE can be used to investigate the performance of important operating system primitives and to determine possible causes for unusually slow system call response times. Finally, we demonstrated the use of SPADE to test a hypothesis about the cause of variability in application-level response time. In the course of developing our system we investigated the suitability of four existing databases for SPADE with respect to their robustness, loading and query performance, user interface, and support for needed datatypes. We discovered that none offered sufficient performance for continuous online updating of the database. Finally, we consider how the concepts in SPADE form the natural foundation for a new class of adaptive, self-tuning systems ideally suited for infrastructure applications.

The use of a database as the central metaphor of SPADE vastly simplifies the process of investigating and analyzing the voluminous and unwieldy data sets generated by a fully-instrumented system. We have demonstrated through example that the ability to issue queries over SPADE's database enables users to easily investigate and correlate performance data gathered from all levels of a system; our experience with our sample application (Apache) only hints at SPADE's potential for guiding users to new insights into the underlying causes of system performance bottlenecks and application performance variability.

12 References

[AS94] Agrawal, R. and R. Srikant. "Fast Algorithms for Mining Association Rules." *Proceedings of the 20th International Conference on Very Large Data Bases*, 487–499. Santiago, Chile, September 1994.

[AP97] Anderson, E. and D. Patterson. "Extensible, scalable monitoring for clusters of computers." In *Proceedings of the 11th Systems Administration Conference (LISA '97)*, 1997.

[ABD+97] Anderson, J., L. Berc, J. Dean *et al.* "Continuous Profiling: Where Have All the Cycles Gone?" In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, 1997.

[AF96] Arpaci, R. and M. Fahndrich. "revEELing Solaris." *UC Berkeley CS294-4 class project*, 1996. http://www.cs.berkeley.edu/~manuel/IRAM/

[BDM99] Banga, G., P. Druschel, J. Mogul. "Resource Containers: A New Facility for Resource Management in Server Systems." In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI99)*, New Orleans, LA, February 1999.

[Bro97] Brown, A. "A Decompositional Approach to Performance Evaluation." Technical Report TR-03-97, Center for Research in Computing Technology, Harvard University, April 1997.

[BOK+99] Brown, A., D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D.A. Patterson. "ISTORE: Introspective Storage for Data-Intensive Network Services." *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[BS97] Brown, A. and M. Seltzer. "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture." *In Proceedings of the 1997 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[CK94] Cmelik, B. and D. Keppel. “Shade: a fast instruction-set simulator for execution profiling.” *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[EWC+96] Endo, Y., Z. Wang, J. Chen, and M. Seltzer, “Using Latency to Evaluate Interactive System Performance,” In *Proceedings of the 2nd OSDI*, Seattle WA, October 1996.

[GKM82] Graham, S., P. Kessler, and M. McKusick. “gprof: A call graph execution profiler.” *SIGPLAN Notices*, 17(6):120–126, June 1982.

[HAR99] Hellerstein, J., R. Avnur, and V. Raman. “Informix under CONTROL: Online Query Processing.” *Unpublished manuscript.*

[Hen99] Hennessy, J. “Back to the Future: Time to Return to Some Longstanding Problems in Computer Systems.” Plenary talk presented at *FCRC ’99*, Atlanta, GA, May 1999.

[LAR93] Larus, J.R. “Efficient Program Tracing,” *IEEE Computer*, 26, 5, May 1993, pp 52-61.

[LS95] Larus, J.R. and E. Schnarr. “EEL: Machine-Independent executable editing.” *CM SIGPLAN PLDI Conference*, June 1995

[LSM99] Lee, W., S. Stolfo, and K. Mok. “A data mining framework for building intrusion detection models.” In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[MIPS90] MIPS Computer Systems. *UMIPS-V Reference Manual (pixie and pixstats).* Sunnyvale, CA, 1990.

[Pat99] Patterson, D.A. “Computers for the Post-PC Era.” Talk presented at UC Berkeley Industrial Relations Conference, February 1999.

[PAB+95] Pu, C., T. Autrey, A. Black, *et al*. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’95)*, December 3-6, 1995, Copper Mountain, Colorado.

[RHW95] Rosenblum, M., S. Herrod, E. Witchel, and A. Gupta. “Complete Computer Simulation: The SimOS Approach.” In *IEEE Parallel and Distributed Technology*, Fall 1995.

[SS97] Seltzer, M. and C. Small. “Self-monitoring and self-adapting operating systems.” *In Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HOTOS-VI)*, 1997.

[SSS96] Seltzer, M., C. Small, and M. Smith. “Symbiotic Systems Software.” In *Proceedings of the Workshop on Compiler Support for Systems Software* (WCSSS ’96).

[Sleepycat] The Berkeley Database (BerkeleyDB), Sleepycat Software, http://www.sleepycat.com.

[SE94] Srivastava, A. and A. Eustace. “ATOM: A System for Building Customized Program Analysis Tools.” Digital Equipment Corporation Western Research Laboratory Research Report 94/2, 1994.

[VTUNE] *VTune: Intel’s visual tuning environment*. http://developer.intel.com/vtune/analyzer/index.htm

Appendix A: Monitoring Database Schema

```
-- This set of SQL commands creates the tables for the monitoring database

create table KMON_LOAD (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        nrunnable       int,    -- number of runnable processes
        PRIMARY KEY (epoch)
);

create table KMON_NETSTATS (
        -- all are reported as diffs from beginning of epoch
        -- sys/netinet/tcp_var.h::struct tcpstat
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        coninitiated    int,    -- connections initiated
        conaccepted     int,    -- connections accepted
        connestablished int,    -- connections established
        conndropped     int,    -- connections dropped
        embconndropped  int,    -- embryonic connections dropped
        connclosed      int,    -- conn. closed (includes drops)
        segstryrtt      int,    -- segs where we tried to get rtt
        timessucceeded  int,    -- times we succeeded
        delayedacksent  int,    -- delayed acks sent
        cdirxmttimeout  int,    -- conn. dropped in rxmt timeout
        retranstimeout  int,    -- retransmit timeouts
        persisttimeout  int,    -- persist timeouts
        kalivetimeout   int,    -- keepalive timeouts
        kprobessent     int,    -- keepalive probes sent
        cdropkalive     int,    -- connections dropped in keepalive
        cdroppersist    int,    -- connections dropped in persist
        cdropmemshrtge  int,    -- connections drained due to memory shortage
        pktssent        int,    -- total packets sent
        datapktssent    int,    -- data packets sent
        databytessent   int,    -- data bytes sent
        datapktsrtrans  int,    -- data packets retransmitted
        databytesrtrans int,    -- data bytes retransmitted
        ackonlysent     int,    -- ack-only packets sent
        wndwprobessent  int,    -- window probes sent
        pktsurgonly     int,    -- packets sent with URG only
        wndwupdonlypkts int,    -- window update-only packets sent
        ctrlpktssent    int,    -- control (SYN|FIN|RST) packets sent
        pktsrcvd        int,    -- total packets received
        pktsrcvdinseq   int,    -- packets received in sequence
        bytesrcvdinseq  int,    -- bytes received in sequence
        pktsrcvdckerr   int,    -- packets received with ccksum errs
        pktsrcvdbadoff  int,    -- packets received with bad offset
        pktsdropnomem   int,    -- packets dropped for lack of memory
        pktsrcvdtooshrt int,    -- packets received too short
        duponlyprcvd    int,    -- duplicate-only packets received
        duponlybrcvd    int,    -- duplicate-only bytes received
        pktswdupdata    int,    -- packets with some duplicate data
        dupbytesinpdp   int,    -- dup. bytes in part-dup. packets
        ooopktsrcvd     int,    -- out-of-order packets received
        ooobytesrcvd    int,    -- out-of-order bytes received
        pktswithdaw     int,    -- packets with data after window
        bytesrcvdaw     int,    -- bytes rcvd after window
```

```
        pktsrcvdaftrcls int,    -- packets rcvd after "close"
        rcvdwndprobepkt int,    -- rcvd window probe packets
        rcvddupacks     int,    -- rcvd duplicate acks
        rcvdacksunsent  int,    -- rcvd acks for unsent data
        rcvdackpkts     int,    -- rcvd ack packets
        bytesackbyrcvda int,    -- bytes acked by rcvd acks
        rcvdwndwupdpkts int,    -- rcvd window update packets
        segdroppaws     int,    -- segments dropped due to PAWS
        hdrpredokack    int,    -- times hdr predict ok for acks
        hdrpredokdata   int,    -- times hdr predict ok for data pkts
        inppktnopcbhash int,    -- input packets missing pcb hash
        tcpnosockonport int,    -- no socket on port
        rcvdacknosyncmp int,    -- received ack for which we have no SYN in compressed
state
        -- these statistics deal with the SYN cache
        entriesadded    int,    -- # of entries added
        connscompleted  int,    -- # of connections completed
        entriestimeout  int,    -- # of entries timed out
        droppedovflw    int,    -- # dropped due to overflow
        droppedrst      int,    -- # dropped due to RST
        droppedicmpunr  int,    -- # dropped due to ICMP unreach
        droppedbucket   int,    -- # dropped due to bucket overflow
        nentriesabrtmem int,    -- # of entries aborted (no mem)
        dupsynrcvd      int,    -- # of duplicate SYNs received
        synsdropped     int,    -- # of SYNs dropped (no route/mem)
        hashcollisions  int,    -- # of hash collisions
        -- sys/netinet/udp_var.h
        -- input statistics
        totinputpkts    int,    -- total input packets
        pktshortrthanhd int,    -- packet shorter than header
        cksumerror      int,    -- checksum error
        datalengrtrpkt  int,    -- data length larger than packet
        udpnosockonport int,    -- no socket on port
        arrasbroadcast  int,    -- of above, arrived as broadcast
        notdelivered    int,    -- not delivered, input socket full
        inpktnopcbhash  int,    -- input packets missing pcb hash
        -- output statistics
        totoutputpkts   int,    -- total output packets
        -- sys/netinet/ip_var.h
        totpktsrcvd     int,    -- total packets received
        cksumbad        int,    -- checksum bad
        pkttooshort     int,    -- packet too short
        notenoughdata   int,    -- not enough data
        iphdrlenltdata  int,    -- ip header length < data size
        iplenltiphdrlen int,    -- ip length < ip header length
        fragsrecvd      int,    -- fragments received
        fragsdropped    int,    -- frags dropped (dups, out of space)
        fragstimedout   int,    -- fragments timed out
        pktsforwarded   int,    -- packets forwarded
        pktsrcvddeunrch int,    -- packets rcvd for unreachable dest
        pktsfwdsamenet  int,    -- packets forwarded on same net
        unkunsupproto   int,    -- unknown or unsupported protocol
        dgramdeltoupper int,    -- datagrams delivered to upper level
        totalippktsgend int,    -- total ip packets generated here
        lostpkts        int,    -- lost packets due to nobufs, etc.
        pktsreassembok  int,    -- total packets reassembled ok
        dgramokfrag     int,    -- datagrams sucessfully fragmented
        outfragscrtd    int,    -- output fragments created
        nofragflagset   int,    -- don't fragment flag was set, etc.
        errorinoptprocs int,    -- error in option processing
        dscrdnoroute    int,    -- packets discarded due to no route
        ipversnotfour   int,    -- ip version != 4
        rawippktsgend   int,    -- total raw ip packets generated
```

```
        fragbadlength   int,    -- malformed fragments (bad length)
        fragdropnomem   int,    -- frags dropped for lack of memory
        iplentoolong    int,    -- ip length > max ip packet size
        -- sys/netinet/icmp_var.h
        -- statistics related to icmp packets generated
        callstoicmperr  int,    -- # of calls to icmp_error
        noerroldipshrt  int,    -- no error 'cuz old ip too short
        noerroldicmp    int,    -- no error 'cuz old was icmp
        -- statistics related to input messages processed
        icmpcodeoutrng  int,    -- icmp_code out of range
        pktlticmpminlen int,    -- packet < ICMP_MINLEN
        badcksum        int,    -- bad checksum
        boundmismatch   int,    -- calculated bound mismatch
        nresponses      int,    -- number of responses [igmp_var.h follows]
        -- sys/netinet/igmp_var.h
        igmpmsgrcvd     int,    -- total IGMP messages received
        igmpfewbytes    int,    -- received with too few bytes
        igmpbadcksum    int,    -- received with bad checksum
        igmprcvdmemqry  int,    -- received membership queries
        igmpinvalqry    int,    -- received invalid queries
        igmprcvdmemrpt  int,    -- received membership reports
        igmprcvdinvlrpt int,    -- received invalid reports
        igmprcvdourgrp  int,    -- received reports for our groups
        igmpsentmemrpt  int,    -- sent membership reports
        PRIMARY KEY (epoch)
);

create table KMON_DISKSTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        diskname        varchar(8),     -- disk name
        transfers       int8,   -- total # of transfers (this epoch)
        indepseekops    int8,   -- # of independent seek ops (this epoch)
        bytesxferred    int8,   -- # of bytes xferred (this epoch)
        timebusysec     int,    -- total time spent busy (this epoch)
        timebusyusec    int,    -- total time spent busy (this epoch)
        PRIMARY KEY (epoch, diskname)
);

create table KMON_BQUEUESTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        freebufq0       int,    -- number of free buffers of type LOCKED
        freebufq1       int,    -- number of free buffers of type LRU
        freebufq2       int,    -- number of free buffers of type AGE
        freebufq3       int,    -- number of free buffers of type EMPTY
        freespaceq0     int,    -- total free space of type LOCKED
        freespaceq1     int,    -- total free space of type LRU
        freespaceq2     int,    -- total free space of type AGE
        freespaceq3     int,    -- total free space of type EMPTY
        PRIMARY KEY (epoch)
);

create table KMON_BIOEVENTS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        rw              int,    -- 0 == read, 1 == write
```

```
        size            int,    -- xfer size, in bytes
        lblkno          int,    -- logical block number
        lblknodiff      int,    -- logical block seek distance from previous
        pblkno          int,    -- physical block number
        pblknodiff      int,    -- physical block seek distance from previous
        PRIMARY KEY (epoch, uniquehi, uniquelow)
);

create table KMON_SCSIEVENTS(
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        errcode         int,    -- SCSI sense error code
        PRIMARY KEY (epoch, uniquehi, uniquelow)
);

create table KMON_SCHEDSTATS_INSTATE (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        nidle           int,    -- SIDL
        nrun            int,    -- SRUN
        nsleep          int,    -- SSLEEP
        nstop           int,    -- SSTOP
        nzomb           int,    -- SZOMB
        PRIMARY KEY (epoch)
);

create table KMON_SCHEDSTATS_INQ ( -- collecting these stats is broken at the moment
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        qno             int,    -- queue number (0-31)
        number          int,    -- number in that queue
        PRIMARY KEY (epoch, qno)
);

create table KMON_SCHEDEVENTS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        uniquehi        int,    -- low 32 bits of cycles timestamp
        uniquelow       int,    -- high 32 bits of cycles timestamp
        pid             int,    -- pid
        timein          float,  -- timestamp when scheduled in, in seconds
        timeout         float,  -- timestamp when scheduled out, in seconds
        prio            int,    -- process priority
        userprio        int,    -- user priority based on p_cpu and p_nice
        pnice           int,    -- process nice value
        hituser         int8,   -- statclock hits in user mode (this quantum)
        hitsystem       int8,   -- statclock hits in system mode (this quantum)
        hitintr         int8,   -- statclock hits in handling interrupts (this
quantum)
        realtimesec     int,    -- real time, seconds (this quantum)
        realtimeusec    int,    -- real time, microseconds (this quantum)
        cputicks        int,    -- ticks of CPU time (this quantum)
        avgcpticks      int,    -- time averaged value of cpticks
        ppid            int,    -- parent process id
        userusedsec     int,    -- user time used, seconds (this quantum)
        userusedusec    int,    -- user time used, microseconds (this quantum)
        sysusedsec      int,    -- system time used, seconds (this quantum)
```

```
        sysusedusec     int,    -- system time used, microseconds (this quantum)
        maxrss          int,    -- max rss
        sharedtxtmemsz  int,    -- integral shared text memory size
        unshareddatasz  int,    -- integral unshared data size
        unsharedstacksz int,    -- integral unshared stack size
        pgreclaims      int,    -- page reclaims, this quantum
        pgflts          int,    -- page faults, this quantum
        swaps           int,    -- swaps, this quantum
        biops           int,    -- block input operations, this quantum
        boops           int,    -- block output operations, this quantum
        msgssent        int,    -- messages sent, this quantum
        msgsrcvd        int,    -- messages received, this quantum
        sigrcvd         int,    -- signals received, this quantum
        volctxtswx      int,    -- voluntary context switches, this quantum
        involctxtswx    int,    -- involuntary context switches, this quantum
        ruid            int,    -- real user id
        PRIMARY KEY (epoch, uniquehi, uniquelow)
);


create table KMON_VMSTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        queuelen        int,    -- length of run queue
        ndiskwait       int,    -- #jobs in disk wait
        npagewait       int,    -- jobs in page wait
        nsleepcore      int,    -- jobs sleeping in core
        nswprunnable    int,    -- swapped out runnable/short block jobs
        -- remaining stats are all in units of pages
        totalvm         int,    -- total virtual memory
        activevm        int,    -- active virtual memory
        totalrmem       int,    -- total real memory in use
        activermem      int,    -- active real memory
        totalshmem      int,    -- shared virtual memory
        activeshmem     int,    -- active shared virtual memory
        totalshrmem     int,    -- shared real memory
        activeshrmem    int,    -- active shared real memory
        freepgs         int,     -- free memory pages
        PRIMARY KEY (epoch)
);

create table KMON_POSTGRES (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        query           varchar(1023), -- query string
        PRIMARY KEY (epoch)
);

create table KMON_MEMBUCKETSTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        bucketsize      int,    -- size of bucket in bytes
        nused           int,    -- number of allocations in use
        nfree           int,    -- number of free allocations
        nreqs           int,    -- number of requests made for this size buf
        highwater       int,    -- high watermark
        overflow        int,    -- number of allocations beyond high watermark
```

```
        PRIMARY KEY (epoch, bucketsize)
);

create table KMON_MEMTYPESTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        type            varchar(15), -- type as ascii string
        itype           int,    -- itype (integer type)
        nused           int,    -- num objs of this type in use
        memused         int,    -- mem used by this type (KB)
        deltamem        int,    -- change in mem usage over epoch
        maxused         int,    -- maximum number of these objects used
        maxallowed      int,    -- max number of these objects allowed to exist
        nreqs           int,    -- requests made for this type over this epoch
        limblocks       int,    -- #times requests blocked hitting the limit
        kmapblocks      int,    -- #times requests blocked for kernel map
        minsize         int,    -- min size of this type
        maxsize         int,    -- max size of this type
        mediansize      int,    -- median size of this type (not weighted!)
        PRIMARY KEY (epoch, itype)
);

create table KMON_INTRSTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        intr0           int,    -- count for interrupt 0
        intr1           int,    -- count for interrupt 1
        intr2           int,    -- count for interrupt 2
        intr3           int,    -- count for interrupt 3
        intr4           int,    -- count for interrupt 4
        intr5           int,    -- count for interrupt 5
        intr6           int,    -- count for interrupt 6
        intr7           int,    -- count for interrupt 7
        intr8           int,    -- count for interrupt 8
        intr9           int,    -- count for interrupt 9
        intr10          int,    -- count for interrupt 10
        intr11          int,    -- count for interrupt 11
        intr12          int,    -- count for interrupt 12
        intr13          int,    -- count for interrupt 13
        intr14          int,    -- count for interrupt 14
        intr15          int,    -- count for interrupt 15
        intr16          int,    -- count for interrupt 16
        intr17          int,    -- count for interrupt 17
        intr18          int,    -- count for interrupt 18
        intr19          int,    -- count for interrupt 19
        intr20          int,    -- count for interrupt 20
        intr21          int,    -- count for interrupt 21
        intr22          int,    -- count for interrupt 22
        intr23          int,    -- count for interrupt 23
        intr24          int,    -- count for interrupt 24
        intr25          int,    -- count for interrupt 25
        intr26          int,    -- count for interrupt 26
        intr27          int,    -- count for interrupt 27
        intr28          int,    -- count for interrupt 28
        intr29          int,    -- count for interrupt 29
        intr30          int,    -- count for interrupt 30
        intr31          int,    -- count for interrupt 31
        intr0r          int,    -- rate for interrupt 0
        intr1r          int,    -- rate for interrupt 1
```

```
        intr2r          int,    -- rate for interrupt 2
        intr3r          int,    -- rate for interrupt 3
        intr4r          int,    -- rate for interrupt 4
        intr5r          int,    -- rate for interrupt 5
        intr6r          int,    -- rate for interrupt 6
        intr7r          int,    -- rate for interrupt 7
        intr8r          int,    -- rate for interrupt 8
        intr9r          int,    -- rate for interrupt 9
        intr10r         int,    -- rate for interrupt 10
        intr11r         int,    -- rate for interrupt 11
        intr12r         int,    -- rate for interrupt 12
        intr13r         int,    -- rate for interrupt 13
        intr14r         int,    -- rate for interrupt 14
        intr15r         int,    -- rate for interrupt 15
        intr16r         int,    -- rate for interrupt 16
        intr17r         int,    -- rate for interrupt 17
        intr18r         int,    -- rate for interrupt 18
        intr19r         int,    -- rate for interrupt 19
        intr20r         int,    -- rate for interrupt 20
        intr21r         int,    -- rate for interrupt 21
        intr22r         int,    -- rate for interrupt 22
        intr23r         int,    -- rate for interrupt 23
        intr24r         int,    -- rate for interrupt 24
        intr25r         int,    -- rate for interrupt 25
        intr26r         int,    -- rate for interrupt 26
        intr27r         int,    -- rate for interrupt 27
        intr28r         int,    -- rate for interrupt 28
        intr29r         int,    -- rate for interrupt 29
        intr30r         int,    -- rate for interrupt 30
        intr31r         int,    -- rate for interrupt 31
        PRIMARY KEY (epoch)
);

create table KMON_SUMSTATS (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        nctxsw          int,    -- #context switches
        ntraps          int,    -- #calls to trap()
        nsyscalls       int,    -- #syscalls
        nhwintrs        int,    -- #device interrupts
        nswintrs        int,    -- #software interrupts
        nfaults         int,    -- total #faults taken
        vmcachelookups  int,    -- VM object cache lookups
        vmcachehits     int,    -- VM object cache hits
        vmcachehitrate  float,  -- VM object cache hit rate (%, floating-point)
        naddrmemfaults  int,    -- #"address memory" faults
        ncows           int,    -- #COWs
        nswapins        int,    -- #swapins
        nswapouts       int,    -- #swapouts
        npswappedin     int,    -- #pages swapped in
        npswappedout    int,    -- #pages swapped out
        npageins        int,    -- #pageins
        npageouts       int,    -- #pageouts
        nppagedin       int,    -- #pages paged in
        nppagedout      int,    -- #pages paged out
        npfblktransit   int,    -- #pfaults blocked b/c req'd page in transit
        npreactivated   int,    -- #pages reactivated from the free list
        clockhandrevs   int,    -- revolutions of the clock hand
        pagedemonscans  int,    -- scans in pageout daemon
        pfreeddaemon    int,    -- pages freed by daemon
        pfreedprocs     int,    -- pages freed by exiting processes
```

```
        pzfod           int,    -- pages zero-filled-on-demand
        nzfod_created   int,    -- # of zfod pages created (not necc. filled)
        nkernpgs        int,    -- #pages in use by kernel (snapshot)
        freetarget      int,    -- target #of pages to keep free (snapshot)
        minfree         int,    -- min # of pages to keep free (snapshot)
        npfree          int,    -- # of free pages (snapshot)
        npwired         int,    -- # of wired pages (snapshot)
        npactive        int,    -- # of active pages (snapshot)
        inactivetarget  int,    -- target #of pages to keep inactive (snapshot)
        npinactive      int,    -- # of inactive pages (snapshot)
        PRIMARY KEY (epoch)
);

create table KMON_SYSCALL (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        name            varchar(40), -- system call name (ascii string)
        num             int,    -- system call number
        pid             int,    -- pid of process issuing system call
        arg1            int,    -- argument #1 to system call
        arg2            int,    -- argument #2 to system call
        arg3            int,    -- argument #3 to system call
        arg4            int,    -- argument #4 to system call
        rval            int,    -- return value of system call
        errno           int,    -- error (errno) set by system call
        elapsedtime     float,  -- elapsed time for system call in seconds
        PRIMARY KEY(epoch, timehi, timelo)
);

create table KMON_PCHIST (
        epoch           int,    -- epoch
        time            float,  -- timestamp, in seconds
        timehi          int,    -- high 32 bits of time
        timelo          int,    -- low 32 bits of time
        fnname          varchar(40), -- function name (ascii string)
        nticks          int,    -- #profiling ticks accumulated during epoch
        timefrac        float,  -- %time spent in this fn during the epoch (fp)
        PRIMARY KEY(epoch, fnname)
);

create table KMON_APACHE (
        seqno           int,    -- request sequence number
        resptime_c      float,  -- response time for serving request, w/close
        resptime_noc    float,  -- response time for serving request, w/o close
        firstepoch      int,    -- first epoch in which request was active
        lastepoch       int,    -- last epoch in which request was active
        fname           varchar(255), -- requested filename
        PRIMARY KEY(seqno)
);
```