Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06). June 26th-30th 2006, Saint-Malo, France.

Stardust: Tracking Activity in a Distributed Storage System

Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs,
Michael Abd-El-Malek, Julio Lopez, Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

Performance monitoring in most distributed systems provides minimal guidance for tuning, problem diagnosis, and decision making. Stardust is a monitoring infrastructure that replaces traditional performance counters with end-to-end traces of requests and allows for efficient querying of performance metrics. Such traces better inform key administrative performance challenges by enabling, for example, extraction of per-workload, per-resource demand information and per-workload latency graphs. This paper reports on our experience building and using end-to-end tracing as an on-line monitoring tool in a distributed storage system. Using diverse system workloads and scenarios, we show that such fine-grained tracing can be made efficient (less than 6% overhead) and is useful for on- and off-line analysis of system behavior. These experiences make a case for having other systems incorporate such an instrumentation framework.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques,Design studies

General Terms

Design,Performance,Measurement,Management

Keywords

End-to-end tracing, request causal chain, Ursa Minor

1. INTRODUCTION

Performance tuning is a complex issue facing administrators of any large-scale system. For example, in database systems, tuning choices include the creation of indices to speed up data lookup, the layout of data on physical storage devices, and the selection of appropriate buffer cache parameters. In storage systems, tuning choices include selecting the right data distribution and load balancing. In operating systems, tuning choices include determining when to upgrade resources (CPU / RAM / disks), how to best partition physical memory among virtual machines, and determining software modules that may be performance bottlenecks. An administrator must deal with users that complain about system performance, identify the source(s) of each problem, and determine corrective actions.

Tuning a distributed system requires good monitoring infrastructure and tools. Current systems provide little assistance. Most insights they give to the administrator come in the form of hundreds of performance counters that the administrator can try to interpret, analyze, and filter to diagnose performance problems. For example, most modern databases and operating systems come loaded with an array of performance counters [21, 29, 33]. Performance counters, however plentiful, are inadequate for two primary reasons. First, in shared environments, aggregate performance counters do not differentiate between different workloads in the system and give only combined workload measurements. If the administrator is attempting to diagnose an issue with one of several workloads, aggregate counters are not helpful. Second, in a distributed system, performance counters cannot be easily correlated to high-level user observations about throughput and latency. The lack of causality and request flow information makes combining information across components difficult.

These shortcomings have led to systems for which administrators need to be very knowledgeable and, hence, system administration accounts for a large portion of the total cost of ownership [4, 14]. They also push administrators to deploy over-provisioned systems. Yet, over-provisioning is also expensive, and even over-provisioned systems need performance tuning when workloads do not behave as expected or change over time.

Stardust is an infrastructure for collecting and querying *end-to-end traces* in a distributed system. Trace records are logged for each step of a request, from when a request enters the system to when it is complete, including communication across distributed components. The trace records are stored in databases, and queries can be used to extract per-request flow graphs, latencies, and resource demands.

This paper describes the design of Stardust and an implementation inside a large-scale distributed storage system [1]. It discusses the challenges faced in building this infrastructure and the opportunities that arise from having it in the system. The challenges included reducing the overhead the infrastructure places on foreground workloads, reducing the amount of spare resources needed to collect and process the traces generated, and ease of trace analysis. The opportunities include concrete tuning problems we are able to solve using Stardust and other tuning problems we have not yet addressed, but we believe are solvable using this infrastructure. We also discuss the limitations of end-to-end tracing, as a performance monitoring tool, and the kinds of problems it will not solve.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMetrics/Performance'06, June 26–30, 2006, Saint Malo, France.
Copyright 2006 ACM 1-59593-320-4/06/0006 ...$5.00.

2. BACKGROUND AND RELATED WORK

Performance monitoring is important throughout a system's lifetime. In the initial system implementation stages, it helps developers understand inefficiencies in the design or implementation that result in poor performance. In the system deployment stages, it helps administrators identify bottlenecks in the system, predict future bottlenecks, determine useful upgrades, and plan for growth. When users complain, administrators can use the observations from periods of poor performance to help understand the source of the problem. Observations from real systems even drive research into building better system models and capacity planning techniques (much related work collected in [24, 26]). This section further motivates our work and surveys the traditional and state-of-the-art approaches in performance monitoring.

2.1 Three "simple" administrator questions

We use three concrete scenarios to focus the discussion in this section.

More RAM or faster disks?: When money is available to upgrade hardware, administrators must decide how to spend their limited budget. "Should I buy more RAM (for caching) or faster disks?" is a simple example choice and even it is not straightforward to answer. The value of increased cache space is access pattern dependent and, worse, workload mixing can muddy the picture of what limits any given application's performance. When using performance counters, in particular, it is unclear which counters should be consulted to answer this question. In the systems we have observed, none of the performance counters are adequate. For example, consider the counter that keeps track of buffer cache hits and misses. Even if that counter indicates that client A's workload never hits in the cache, it does not mean that adding more RAM for the cache would not help—a workload, for example, that scans a 500 MB object repeatedly, but has been allocated only 499 MB of RAM space (and thus never hits in buffer cache with an LRU replacement policy), would benefit greatly from a 1 MB increase in RAM space. The workload would then see a 100% hit rate. Similarly, consider a counter that keeps track of the average disk queue size. A large value does not necessarily mean that faster disks would be better than more RAM.

Decisions are even more difficult for a shared infrastructure supporting multiple clients. For example, one client may benefit most from a RAM upgrade while another would benefit more from faster disks. Aggregate counters show overall averages, rather than per-workload information, so this information will be hidden. If one client is more important than the other, going with an average-case choice is not appropriate. Interactions among workloads can also create situations where changing the amount of buffer cache for one causes a ripple effect on the performance of the others (e.g., the sequence of disk accesses changes). For example, we have seen cases where improving the cache hit rate for one client also provides a bigger than expected efficiency boost for another by reducing disk-level interference [42].

Where does time go?: When a particular set of requests are slower than expected, an administrator often needs to know why (and then implement a fix). "Where are requests from client A spending most of their time in the system?" is an example administrator question. This question is representative of situations in distributed environments with multiple processing components (e.g., a request passing through a web server which checks a database which retrieves data from a storage system). The administrator may want to know which component accounts for which fraction of the average request's latency. Answering this question requires creating a request's latency graph as the request moves from component to component in the distributed system. Aggregate counters do not help with this. One needs to know how the request moved through the system and how long it spent at each component.

Why is the client complaining?: When users complain about their performance, administrators must figure out what happened, differentiate between transient problems (which can often be ignored) and recurring ones (which should be fixed), and decide what to do. "Why was the application's performance unacceptable at 2pm?" is an example starting point. At a minimum, the system will need to retain performance observations for a period of time so that the administrator can go back and check. But, looking at performance counters, like CPU load, disk I/O rate, buffer cache hits and misses will rarely be sufficient for root-cause analysis for the reasons explained above. As well, the administrator may need to know the specific sequence of requests that led to the poor performance the client experienced and how those requests moved through the distributed system.

2.2 Traditional performance measurement

Traditional performance monitoring consists of pre-defined counters, such as "number of requests completed" and "average queue length". In practice, performance instrumentation is found mainly in single-node systems [9, 21, 27, 29, 33]. There are some monitoring infrastructures designed for distributed systems [5, 25], but they focus on aggregate resource consumption statistics rather than per-client or per-request information. Such aggregate performance monitors provide little assistance with problems like those discussed in Section 2.1.

Although they are useful as a method to visualize overall system performance and component utilizations, performance counters have two primary shortcomings when deeper analysis is needed. First, in an infrastructure shared among multiple clients, performance counters cannot differentiate among their different workloads. They provide only aggregate values that do not help an administrator understand individual workload behavior and demands. For example, in a system with an I/O-bound workload from a high-paying customer and several CPU-bound background workloads, an administrator cannot easily quantify the effect of a hypothetical resource upgrade (e.g., a faster CPU) on the high-paying customer. Counters may indicate that the CPU is a bottleneck, as it indeed may be for the combined workload, but upgrading the CPU may have no effect on the I/O-bound workload.

Second, performance counters cannot be easily correlated to high-level user observations about throughput and latency in a distributed system. In a busy web server with a backend database, is the high CPU utilization at the web server, the high I/O rate at the database server, both, or none of them, responsible for the high latency clients are noticing? Performance counters act like symptom sensors, without enabling root-cause analysis of the problem.

Detailed process accounting systems can address some of the shortcomings of performance counters. For example, Bouhana [8] describes an accounting system that keeps per-user, per-resource demands in order to later bill the user appropriately. Existing accounting systems, however, are limited to centralized systems and simple batch processing systems. In most distributed systems, where request processing may involve applications running on several machines, existing approaches do not work. Among other things, Stardust can be used as an accounting system for distributed systems.

System logs are often used to capture basic workload or utilization information. The Windows operating system, for example, offers the ability to log performance counters and request/reply calls for later analysis [27, 29]. The most common use of such logging is retaining a history of HTTP calls or SQL database queries. Such

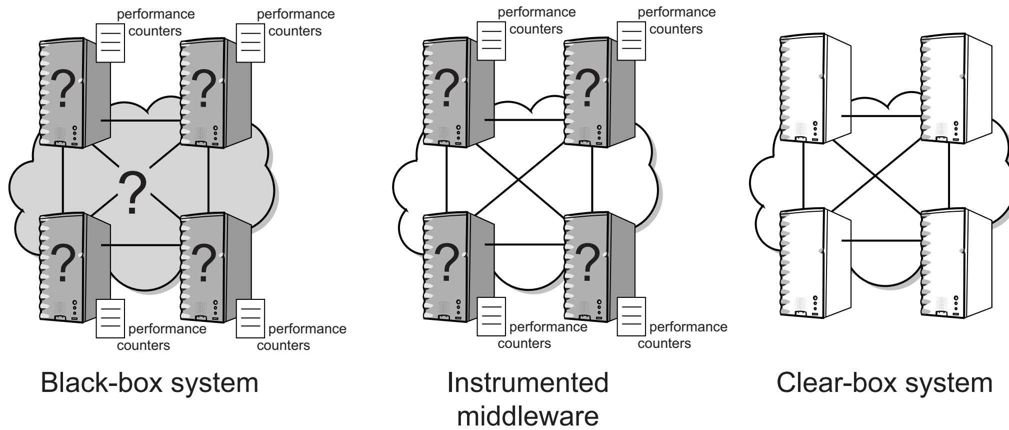


Figure 1: The instrumentation framework depends on the system under consideration. Black-box systems are made of components that work together through well-defined interfaces but are closed-source. Instrumented middleware systems consist of the black box components running on top of a well-known middleware that provides resource multiplexing, management and accounting. Clear-box systems are a term we use for systems whose internals are completely known, either because the system is being built from the start or because its source code is available. Such systems offer the opportunity to have the necessary instrumentation built-in from the start.

logging is useful for workload analysis and even trace-based simulation but, as commonly configured, provides little help with performance analysis. Such logs are usually coarse-grained, and there are no integrated tools that allow the administrator to correlate log entries with system performance information. Stardust builds on fine-grained logging/tracing of activity across a distributed system, enabling both traditional information extraction and detailed performance analysis.

2.3 Recent work

Researchers are now exploring the use of *end-to-end tracing* of requests in a distributed system to better inform diagnosis and tuning questions. End-to-end tracing refers to collection, storage, and correlation of activity records that are generated by a single request from the moment it enters the first node in the distributed system until it leaves the system. Research in this area has focused on three system types (illustrated in Figure 1): black-box, middleware-based, and clear-box systems.

Black-box systems are constructed of components that work together through well-defined interfaces but are closed-source. Although only high-level information can be determined in such an environment, researchers are developing approaches to determine causal paths of requests and the relationship between individual component performance and overall performance. For example, Aguilera et al. [2] have shown that coarse-grained end-to-end traces can be extracted via passive network monitoring without requiring any legacy application changes. Further, they showed that such tracing is sufficient to identify black-box components that are bottlenecks and guide an administrator's focus to them. Cohen et al. [11] explore the efficacy of statistical correlating the values of per-black-box performance counters with high-level user observations, such as throughput or latency, to identify relationships that can guide diagnosis. These approaches improve on traditional approaches and can work when otherwise uninstrumented third-party applications are utilized.

Instrumented middleware systems are often deployed as a set of black box components running on top of middleware, such as J2EE or .NET, that provides resource multiplexing and management [28, 30]. Systems such as Pinpoint [10] tag requests as they flow through the J2EE middleware and correlate middleware measurements with application-level throughput and latency. Xaffire [44] and Application Assurance [35] are commercial products that use similar tagging techniques. Such a model provides deeper insight than pure black box and traditional approaches, but still leaves intra-component resource usage and delay sources unclear.

We use the term *clear-box system* to describe a system whose internals can be modified and understood, either because the system is being built from the start or because its source code is available. Such systems offer the opportunity to have fine-grain instrumentation built in. For example, Magpie is a prototype system that collects traces at different points in a system and creates causal paths from those traces [7, 22]. Magpie relies on programmers to place instrumentation points in the appropriate system modules. In return, it offers critical path analysis and per-workload, per-resource monitoring. ETE is a similar system that is used to measure end-to-end response times [18]. Hrischuk et al. define a specialized language to describe end-to-end traces and measure per-workload and per-resource demand as well as request response times [20]. Stardust and this paper build on these ideas and takes them further by developing an efficient querying framework for traces, reporting experiences from use in a real large-scale system, and performing feasibility studies under various system loads.

3. STARDUST'S DESIGN

Stardust's design was motivated by several goals:

Resource usage accounting: The instrumentation framework must provide accurate aggregate and per-client resource accounting. Aggregate accounting is sufficient when the administrator is concerned with the load on a resource. But, per-client accounting is needed to understand how individual clients contribute to that load. In a distributed system, a request may propagate through several machines, requiring per-client accounting of all resources used in each machine. Resources of interest in a storage system include the CPU, buffer cache, network and disks.

Request latency accounting: The instrumentation framework must provide per-client request latency information that records where a request spends its time as it flows through the system. Different clients may have different latency profiles. A client whose requests hit in the buffer cache, for example, will have a different profile than a client whose requests miss. A request may span machines in a distributed system, so care must be taken to causally

link together sub-requests in each machine that belong to the same original request.

Instrumentation framework efficiency: The instrumentation framework should interfere minimally with the workloads running in the system. We envision the framework to be monitoring the system at all times; hence, overheads must be minimal. In addition, the programming burden for implementing the framework inside a system should be low.

Querying efficiency: The instrumentation framework must provide a flexible query interface. In a distributed storage system with hundreds of nodes and clients, it is important to have a versatile way to query the wealth of information generated.

3.1 Activity tracking and querying

Stardust is made of two components: the activity tracking infrastructure (ATI) and the querying infrastructure. The ATI is responsible for tracking every client request along its execution path. The ATI retains activity records, such as buffer cache reference records, I/O records, and network transmit/receive records. The sequence of records allows tracking of a request as it moves in the system from one computer, through the network, to another computer, and back.

An activity record is a sequence of values related to a record type. Figure 2 shows an example activity record. Each activity record contains an automatically-generated header comprised of a timestamp, breadcrumb, kernel-level process ID, and user-level thread ID. Each timestamp is a unique value generated by the CPU cycle counter that permits accurate timing measurements of requests. The breadcrumb permits records associated with a given request to be correlated within and across computers. Activity records are posted at strategic locations in the code so that the demand on a resource is captured. These locations are often the point of arrival to and departure from a processing queue. For example, the disk activity record is posted both when the request is sent to disk and when the request completes. Both postings contain the same breadcrumb, because they belong to the same request, and so can be correlated. Records are posted on the critical path; however, as our evaluation shows, such posting causes minimal impact on foreground performance.

Each computer runs a single ATI instance. An ATI instance is responsible for presenting any process running on that computer with APIs for posting and querying activity records. For querying flexibility, ATI records are stored in relational databases (Activity DBs). Activity records posted to an ATI instance are periodically sent to Activity DBs. Activity DBs run on the same infrastructure computers with the rest of the system. The DBs store the records in relational tables and answer queries on those records. Storing activity records in a database allows a flexible querying interface.

Activity DBs are part of the querying infrastructure and they can be *queried* using SQL. For example, to get a disk I/O trace for a certain storage-node, one could query the Activity DB that keeps records for that storage-node's disk activity records. Activity records are effectively a super-set of performance counters. Any performance counter value of interest can be extracted by querying the Activity DBs.

3.2 Resource usage accounting

This section describes how the ATI enables the extraction of the per-workload demand placed on four common storage system resources: CPU, buffer cache, network and disk. When requests enter the system, they are tagged with the user's ID. The first component also assigns an initial breadcrumb. This breadcrumb is passed between components as the request is serviced.

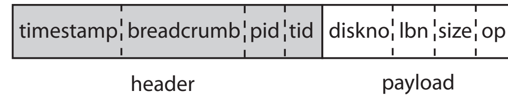


Figure 2: Example activity record. Each activity record has a common header and a payload. The payload for the disk request activity record shown includes the disk ID, logical block number (LBN), size of the I/O in bytes, and operation type.

CPU demand: To measure per-client CPU demand, a component must post activity records related to thread context switching. Context switching happens both in preemptive systems (when it is time to run another thread) and non-preemptive systems (when a thread yields). Thus, CPU processing of a request may be suspended and resumed several times By monitoring the context switches and the requests being processed during thread run time, Stardust charges a request the exact amount of CPU time it used. That time is the sum of the time any thread spent processing that request and any of its sub-requests.

Buffer cache usage: Buffer cache usage for an individual client is measured by posting activity records each time the buffer cache is accessed. Accesses include read hits and misses, writes, readaheads and evictions. The buffer cache demand for a user during any time period T is determined by the sequence of such accesses.

Network demand: Network demand for an individual user is measured by posting a *NetworkTransmit* activity record each time a request is transmitted from one component to another. These records contain, among other attributes, the number of bytes sent from the source to the destination component. The demand is then the total number of bytes transmitted during any time period T.

Disk demand: Disk demand for an individual client is measured by posting a *DiskOp* activity record each time a request initiates a disk request. An operation denotes the beginning or completion of a read or write. The disk demand during any time T is the sum of disk service times for the user's requests.

3.2.1 Measuring delayed demands

There are two important, and tricky, cases that complicate resource demand accounting. First, whereas read requests are usually synchronous (the user has to wait until the read completes before proceeding), there are asynchronous requests (e.g., writes). A write will often be inserted into a cache and have control returned to the user. The write propagates to other components (e.g., disk) at a later time. This is often done to hide the latency of writes and results in significant performance gains. Resource accounting for those requests must occur, however, even after control is returned to the user. Second, some requests from different users may be coalesced into a single, larger request to improve performance (e.g., coalescing disk requests). It is important to bill the resource usage of this larger request to the proper original requests.

Figure 3 shows both these problems by illustrating the typical path of a write request. The request arrives in the system (denoted by the first black node) and departs (the second black node) after it has been stored in cache. At a later time, denoted by the broken arrow (depicting the first problem), the request is coalesced with other requests and sent to the storage-node (depicting the second problem). The storage-node, in turn may split it into sub-requests. For accounting purposes it is important to capture these cases, especially because writes are frequent in a storage system.

Stardust solves the first issue by storing the breadcrumb as part of the data in the cache. When the request is later processed, any sub-requests it generates use that breadcrumb and thus the original request is properly billed. If that request is coalesced with other

requests, the many-to-one relationship is noted (through an explicit "stitch" record), and any resources the larger request subsequently uses are billed to each original request proportionally.

3.3 Request latency accounting

This section describes how Stardust provides per-client request latency information that shows where a request spends its time as it is processed in the system. Each instrumentation point can be considered as a node in a *latency graph*, with links between nodes denoting causal relationships. These links also capture latency information between the instrumentation points. As a simple example, if one instrumentation point was before a disk request and another after the request completed, the link between them would denote the disk latency.

Identifying causal paths in a distributed environment: Requests often generate multiple activity records at different components in a distributed system. The components of a distributed system are not expected to have synchronized clocks. It is important to causally link, or *stitch*, activity records together so that the path of the original request can be reconstructed in such environments. On the same machine, two records R_1 and R_2 are totally ordered by their timestamp. If R_1 and R_2 have the same breadcrumb, but R_1 happens before R_2 then R_1 is a parent of R_2. On different machines, such ordering is possible only if the records R_1 and R_2 (which are created on their respective machines) are explicitly related through a "stitch" record that contains the breadcrumb of R_1 (or any of its children' sub-requests) and the breadcrumb of R_2 (or any of its parents' requests).

Figure 3 shows an example path to visually explain the above rules. Two physical machines are connected through a network. In a distributed storage system, the first machine could be a generic Metadata Server (which keeps information on where data is located) and the second a generic Storage-Node (which actually stores the data). Nodes in this illustration show activity records. Links between records show nodes that are related because they originated from the same breadcrumb (i.e., breadcrumb 10). On each machine, all records are totally ordered by a local timestamp, denoted by the timeline at the bottom. To show that all records on the first machine happened before any records on the second machine a "stitch" record of the form $(bc = 11, bc = 12)$ is posted just before the request leaves the first machine. The stitch record links the last record on the first machine (child record of the originating request) to the first record on the other machine (parent record for all subsequent records on that machine).

3.4 Instrumentation framework efficiency

There is overhead associated with ATI traces. However, the overhead can be made negligible in most cases. CPU overhead at machines posting events is kept to a minimum by reducing the ATI client work. The client only has to encode the trace records to network byte order and place them in pre-allocated buffers. When the buffers fill up, records are sent to the Activity DBs. Thus, we are trading off the possibility of partial trace loss due to machine failure with lower tracing overhead.

Network and storage overheads are closely related; both depend on the quantity of trace data generated. Ultimately, the quantity of trace data generated depends on intended usage. Section 6 describes several concrete experiences we have had in solving real problems using such traces. In general, performance problems that can be answered by looking at resource load require only per-client, per-resource performance counters. For such problems, it is possible to drastically reduce the amount of traces kept. This is done by transforming the raw traces into per-workload, per-resource demand numbers, as well as per-workload graphs that shows the latency of requests as they pass through different modules, every time period T. Section 5 quantifies the space reduction from pruning.

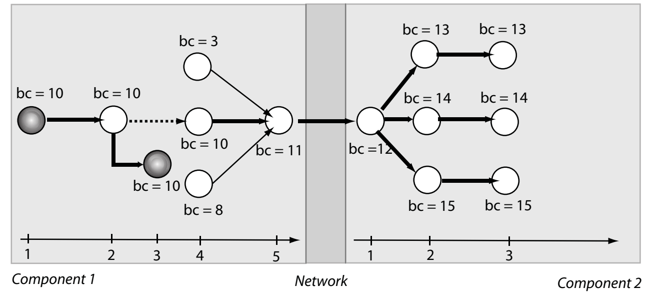


Figure 3: Example causal path created by a generic Write call. This diagram shows the creation of a causal path in a distributed environment with two components. Nodes in this illustration show instrumentation points in the system, and links show a request propagating through two such points. This diagram also illustrates the case when a Write request is declared complete (component 1, time 3) before it is propagated to the second component and when multiple writes are coalesced (component 1, time 5).

There are, however, several performance tuning problems that require certain types of records to be kept in their raw form. Section 5 analyses the feasibility of keeping full traces and the efficiency of pruning them. Section 6 then discusses several concrete uses of these traces and provides guidelines to help determine the right amount of traces to be kept for solving a certain problem. In general, we observe that a storage system sustains a limited number of operations a second (several thousand). This allows the amount of trace records to be relatively small.

3.5 Querying framework efficiency

Activity records are stored in Activity DBs, which use relational tables. Any internal system entity (or external administrator) can use SQL to analyze traces. Each Activity DB contains all the trace records associated with a set of clients and machines. Thus, no query will involve accessing more than one Activity DB. We considered using distributed databases, but opted for simplicity.

Each Activity DB stores activity records in a number of tables, one for each record type. The system is designed such that the Activity DBs do not need to be restarted or recompiled if a component posts new record types. New tables can be created on-the-fly based on an XML description of the new record type.

One can generate any performance counter from the end-to-end activity traces. We call the performance counters generated from the traces *virtual* performance counters because they are not hard-coded into the code, but can be generated on the fly using SQL queries on the trace tables. As a simple example, consider the traditional counter that keeps track of the number of hits in a component's buffer cache (e.g., on a Storage-Node component). In our system, that counter is generated from the following SQL query on the table that holds the buffer cache records for that Storage-Node:

SELECT $count(*)$
FROM $STORAGE_NODE_BUFFER_CACHE_TABLE$
WHERE $optype = BUFFER_READ_HIT$

In general, we have found that using a common querying language like SQL allows flexibility in querying.

4. IMPLEMENTATION IN URSA MINOR

We have designed and implemented a storage system, Ursa Minor, to target research problems in system management. Ursa Mi-

	Record Type	Arguments	Description
CPU demand	*UserThreadSwitch*	*oldthread, newthread*	A user-level context switch
	KernelProcessSwitch	*CPU ID, oldprocess, newprocess*	A kernel-level context switch
Buffer cache demand	*BufferReadHit*	*file, offset, size*	Denotes a buffer cache hit
	BufferReadMiss	*file, offset, size*	Denotes a buffer cache miss
	BufferWrite	*file, offset, size*	Denotes a write and marks buffer dirty
	BufferReadAhead	*file, offset, numpages, pagesize*	Prefetch pages (non-blocking)
	BufferFlush	*file, offset, size*	Flush a dirty page to disk
	BufferEvict	*file, offset, size*	Evict a page from the cache
Network demand	*NetworkTransmit*	*sender, receiver, numbytes*	Monitors network flow
Disk demand	*DiskOp*	*disk ID, LBN, size, operation*	Monitors disk activity

Table 1: Activity records used to measure resource consumption. *KernelProcessSwitch* records are provided by the Linux kernel (other operating systems, such as Windows, already expose kernel-level context switches [27]); the remainder are posted from instrumentation points in user-level processes. Note that there are multiple buffer caches in the system (e.g., at client, metadata service and storage-nodes), hence the buffer cache records are posted at all those levels.

nor is designed from a clean slate; hence, we had the opportunity to include the instrumentation in the design of the system from the beginning without the need to retrofit an existing system.

The goals of this system are described by Ganger et al. [13] and the architecture and implementation are described by Abd-El-Malek et al. [1]. At the core of the architecture is the separation of mechanical functions (servicing client requests) from managerial functions (automating administrative activities). The managerial tier consists of agents and algorithms for automating internal decisions and helping administrators understand the consequences of external ones. The mechanical tier is designed to self-monitor, through Stardust, and also includes self-predictive capabilities used by the management tier, as described in [42]. Below we define the main structural components of the system.

Clients: Clients of the system access data. Data may have different availability, confidentiality and performance goals. Clients make use of the PASIS *protocol family* to encode data [15, 43]. For the purposes of this paper we only consider data replication, although several other schemes to meet the above goals are possible. Currently, clients in our setting use the NFS data access protocol [41]. Clients contact NFS servers which in turn read and write data to the storage-nodes on behalf of the clients.

Storage-nodes: The storage-nodes have CPUs, buffer cache and disks. Storage-nodes are expected to be heterogeneous, as they get upgraded or retired over time and sometimes are purchased from different vendors.

Metadata service: The metadata service (or MDS) is responsible for keeping track of files in the system. It is also responsible for client access control through the use of capabilities. An access to a file usually is preceded by an access to the MDS to get the metadata for accessing that file. Once the metadata is obtained, the client interacts directly to the storage-nodes to access the data.

Other services: There are several other services that run on the same machines as the storage-nodes and share resources with them. Such services include Stardust and various other services that perform background maintenance work.

Figure 4 shows a typical request flow through the distributed system. There are several resources used by a request as it flows through the system. First, CPU computation is required at both the client and at the storage nodes. The client requires CPU computation to encode and decode the data into *fragments* that are stored onto N storage nodes, for N-way replication. Part of encoding may be compressing or encrypting the data. The storage nodes require CPU to check the integrity of data blocks, through checksums. Second, buffer cache is required at the client, metadata service and

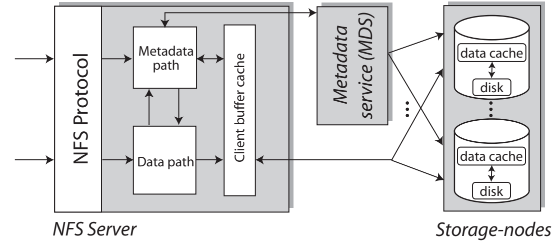


Figure 4: Typical request path through the system. A request from a client enters the storage system at the NFS server. The server consults the metadata service to get the metadata from the client. Once the metadata is received, the data request is served through the data path. The request may hit in the NFS server cache or miss and have to be serviced from the storage-nodes. The request may be sent to more than one server, for example, when the data is replicated or striped among many nodes.

at the storage nodes to hold frequently accessed data or metadata. Third, network bandwidth is required to transmit requests and responses from the various components in the system. Fourth, disk bandwidth is required at the storage nodes to process read requests that miss in the cache and write requests.

We changed the RPC layer in Ursa Minor to have scheduling support for the ATI traces. Without scheduling support, large buffers of traces are periodically sent to the Activity DBs. Any foreground requests that are blocked behind the buffers incur a large latency. Scheduling support mitigates this effect. With scheduling support, the activity traces buffers are split into smaller sizes, and each small chunk is given a lower scheduling priority than foreground requests. This greatly reduces the additional latency seen by foreground requests, as Section 5 quantifies.

4.1 Instrumentation points

Table 1 shows the records used to measure resource demands. Table 2 shows the records used to measure latencies. Some records are used for both. There are approximately 200 instrumentation points in our system, which currently has over 250,000 lines of code. Almost all instrumentation points are posted from user-level processes, because most request processing in Ursa Minor is done in user-level threads. The only exceptions are the kernel-level context switch records (*KernelProcessSwitch*), which are posted by the Linux kernel. This was the only modification necessary to the operating system. User-level context switches (*UserThreadSwitch*) are posted from the State Threads library [39].

In Ursa Minor some level of monitoring is performed at all times.

	Record Type	Arguments	Description
NFS service	*NFSCall_type*	*user ID, call args*	Request arrives at the NFS service
	Buffer_type	*buffer args. See Table 1*	Request accesses the NFS buffer cache
	NFSReply_type	*reply args*	Request exits from the NFS service
MDS service	*MDSCall_type*	*call args*	Request arrives at the MDS service
	Buffer_type	*buffer args. See Table 1*	Request accesses the MDS buffer cache
	MDSReply_type	*reply args*	Request exits from the MDS service
Storage-node	*S-NCall_type*	*call args*	Request arrives at the storage-node service
	Buffer_type	*buffer args. See Table 1*	Request accesses the storage-node buffer cache
	S-NReply_type	*reply args*	Request exits from the storage service
	DiskOpCall	*call args*	Request accesses the storage-node's disk service
	DiskOpReply	*call args*	Request exits from the storage-node's disk service

Table 2: Activity records used to measure request latency. The above records capture entrance and exit points for key services in the system. NFS calls monitored include most calls specified in the NFS protocol [41], of which the most common are : NFS_GETATTR, NFS_SETATTR, NFS_LOOKUP, NFS_READ, NFS_WRITE, NFS_CREATE, NFS_MKDIR, NFS_REMOVE, NFS_RMDIR, NFS_RENAME and NFS_COMMIT. MDS calls monitored include: MDS_LOOKUP, MDS_CREATE_OBJECT, MDS_RELEASE_OBJECT, MDS_APPROVE_WRITE, MDS_FINISH_WRITE. Storage-node calls monitored include: READ, WRITE, CREATE, DELETE. Disk calls monitored include: READ, WRITE.

It includes all the above record types. Further types can be added by programmers through new releases of the system. Such record types may be enabled or disabled at run time. That encourages programmers to insert as many record types as necessary to diagnose a problem; they can always turn them off by default and re-enable them when a product in the field needs to be analyzed. We currently use a small embedded database, SQLite [40], for the ActivityDBs.

5. EVALUATION

This section evaluates the efficiency of the instrumentation and querying framework. Section 6 illustrates experiences we have had with the system.

5.1 Experimental setup

In the following experiments, we used Dell PowerEdge 650 machines equipped with a single 2.66 GHz Pentium 4 processor, 1 GB of RAM, and two Seagate ST33607LW (36 GB, 10k rpm) SCSI disks. The network configuration consisted of a single Intel 82546 gigabit Ethernet adapter in each machine, connected via a Dell PowerConnect 5224 switch. The machines ran the Debian "testing" distribution and used Linux kernel version 2.4.22. The same machine type was used both for clients and storage-nodes. The storage-nodes used one of the two local disks for data; the other contained the operating system. Each NFS server and storage-node has 256 MB and 512 MB of buffer cache respectively. The experiments used several workloads with varying characteristics to assess the efficiency and efficacy of Stardust.

OLTP workload: The OLTP workload mimics an on-line database performing transaction processing. Transactions invoke 8 KB read-modify-write operations to a small number of records in a 5 GB database. The performance of this workload is reported in transactions per minute (tpm).

Postmark workload: Postmark is a file system benchmark designed to emulate small file workloads such as e-mail and netnews. It measures the number of transactions per second that the system is capable of supporting [23]. A transaction is either a file create or file delete, paired with either a read or an append. The configuration parameters used were 100000 files, 50000 transactions, and 224 subdirectories. All other parameters were left as default. Postmark's performance is reported in transactions per second (tps).

IOzone workload: IOzone is a general file system benchmark that can be used to measure streaming data access (e.g., for data

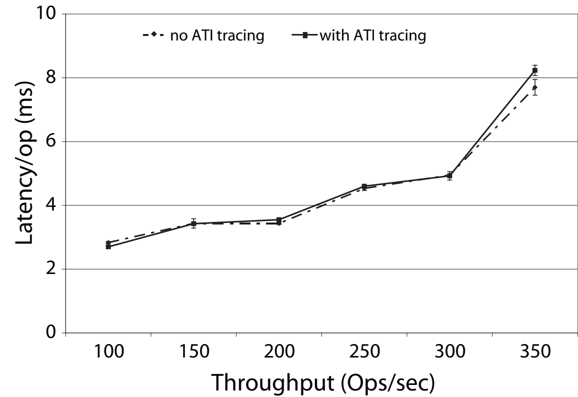


Figure 5: Impact of instrumentation on latency of SFS workload. The average and standard deviation of five runs is shown. For all throughput and latency levels the overhead of instrumentation is low.

mining) [32]. For our experiments, IOzone measures the performance of 64 KB sequential writes and reads to a single 2 GB file. IOzone's performance is reported in megabytes per second read.

"Linux build" development workload: The "Linux build" workload measures the amount of time to clean and build the source tree of Linux kernel 2.6.13-4. The benchmark copies the source tree onto a target system, then cleans and builds it. The results provide an indication of storage system performance for a programming and development workload. The source tree consumes approximately 200 MB. The performance of this workload is measured in seconds to complete the build process.

SPEC SFS Suite: SPEC SFS is a benchmark suite designed to measure the response time of NFS requests for varying throughput levels. The latency of the NFS server is measured as a function of throughput.

"S1-V0" scientific workload: This workload corresponds to storing the output from sample queries performed by a system designed to analyze multi-dimensional *wavefield* datasets. Query outputs are written to storage for further processing. In particular, S1 and V0 correspond to sample queries on multi-gigabyte seismic wave-fields produced by numerical simulations of ground motion wave propagation during strong earthquakes in Southern California [3]. S1 corresponds to the output of temporal queries on a plane,

	CPU demand	Network and storage demand (MB/s)	Performance without tracing	Performance with tracing
Postmark	0.9%	0.34	11 tps	11 tps
OLTP	0.7%	0.57	910 tpm	898 tpm
IOzone	0.1%	3.37	38 MB/s	36 MB/s
Linux-build	0.1%	0.49	1094 secs	1101 secs
S1-V0	0.8%	1.85	669 secs	686 secs

Table 3: Macro-benchmark performance. This table illustrates the overheads of the ATI. The ATI places demands on the CPU for encoding and decoding trace records, network and storage for sending the traces to the Activity DBs and storing them. It also places a fixed demand of 20 MB of buffer cache at each client machine. The impact of the instrumentation on the workload's performance is less than 6%.

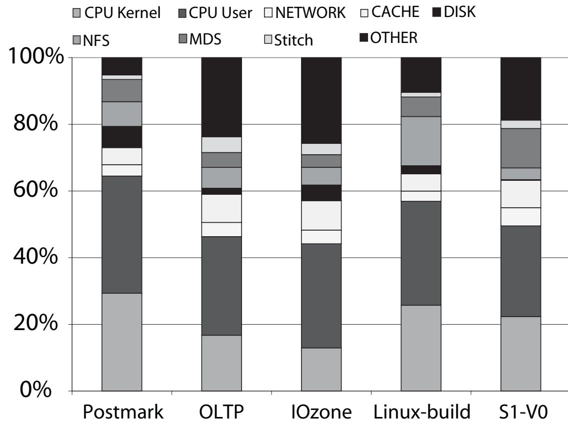


Figure 6: Source of trace records. The largest amount of traces comes from recording every context switch in the system, in order to measure CPU demand per workload. The least amount of traces comes from the I/O subsystem, since many of the workloads fit in buffer cache. Included in the "Other" category are various database-specific overheads and a global index that keep track of the various tables each request uses.

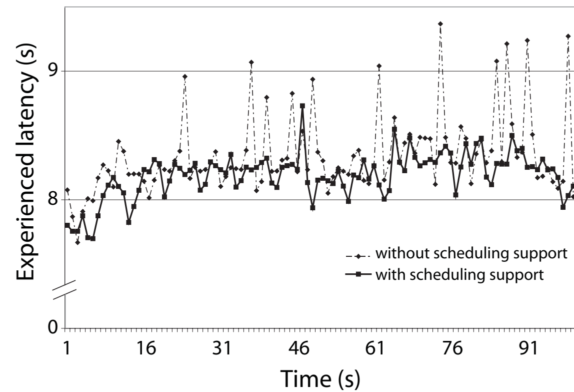


Figure 7: Network scheduling support for the ATI. Latency as seen by Postmark requests, with and without scheduling support.

and V0 corresponds to the output of temporal queries on a volume. The performance of this workload is measured as the overall run time for each workload.

All experiments are run five times and the average is reported, unless otherwise mentioned.

5.2 Instrumentation framework efficiency

Baseline experiments: Table 3 shows the overheads of the end-to-end tracing when each of these workloads is run in isolation. The application data was stored on a single storage-node (i.e., there is no data replication). There is a single Activity DB for the traces. The baseline performance and performance with tracing enabled is shown. As seen from the table, the ATI demands on the CPU, network, and storage are relatively small. The instrumentation added at most a 6% performance penalty. Figure 5 shows the output from the SPEC SFS benchmark. Throughout the curve (from low to high NFS server load) the impact of the instrumentation is low.

Figure 6 shows the source of the traces. For all workloads, most of the tracing data comes from collecting CPU context switches. Each time a context switch (at the kernel or user level) occurs, it is logged ("CPU Kernel" and "CPU User" categories). Many of the workloads fit fully in buffer cache, and only a few of them generate disk I/O records. A considerable amount of tracing overhead comes from keeping causal path information in the form of "stitch" records.

Network scheduling support for traces: Figure 7 shows the impact of adding network scheduling support for the activity traces. We modified the RPC layer in Ursa Minor to have scheduling support for the ATI traces. Without scheduling support, large buffers of traces are periodically sent to the Activity DBs. Any foreground requests that are blocked behind the buffers incur a large latency. Scheduling support mitigates this effect. With scheduling support, the activity traces buffers are split into smaller sizes, and each small chunk is given a lower scheduling priority than foreground requests. This greatly reduces the additional latency seen by foreground requests.

Efficiency with of data redundancy: Because Ursa Minor is built on commodity components, to ensure an acceptable level of reliability and crash tolerance, data is often replicated across storage-nodes. N-way replication refers to data being replicated across N storage nodes. During a write, data is written to all N nodes, and during a read, data is read from one of the N storage nodes. Figure 8 shows the overhead of tracing when replication is used for each of the representative workloads. In general, the overhead increases linearly with the replication degree, since as the replication factor increases more storage nodes are accessed. The overhead increases for several reasons. First, there is more network demand, since more storage-nodes are accessed (during writes). Second, there is more buffer cache demand, since now a data block resides on N nodes. Third, there is more disk demand, since a data block is written on N disks. In general, the amount of trace records increases as replication increases and the trend is similar for all workloads.

5.3 Querying framework efficiency

In addition to resource overheads, an important property of Stardust is ease of querying. In particular, creating a causal path for a request is a common operation that needs to be efficient. Table 4 shows the average number of SQL queries required to create such a path for each of the workloads (these queries are issued internally). Providing the number of queries is more useful than providing the

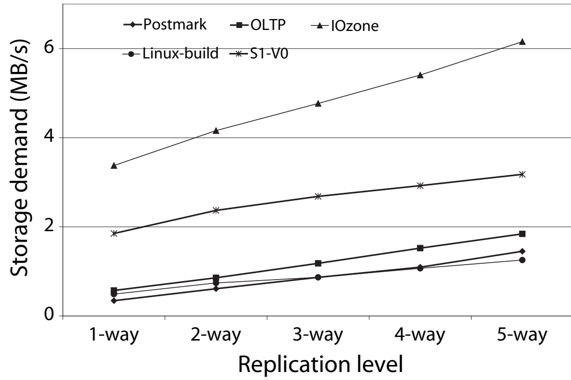


Figure 8: Amount of traces coming as a function of replication level. Changing the replication level results in more storage-nodes being used for a given workload. Each of the storage-nodes generates a certain amount of CPU, network, buffer cache and disk traces.

	Read path	Write path
Postmark	8.5	19
OLTP	8.5	135
IOzone	11.4	109
Linux-build	8.2	42.9
S1-V0	4	12.5

Table 4: Number of SQL queries for creating request path. 10000 causal paths are created and averaged for each workload.

time required to create the path, since that time depends on factors such as the database used, amount of buffer cache dedicated to the database, whether the database if over a network link or is local. The number of SQL queries provides a more standard way to compare the work needed to create a path. All queries are simple and do not involve joins. The column searched is the breadcrumb column.

The time to create a path depends on two main workload factors. First, the deeper a request flows into the system (e.g., when it misses in the NFS server cache and has to go to the storage-nodes) the longer it takes to re-create its path. Writes, for example, tend to have deeper paths than reads, since reads often hit in the buffer cache. Second, coalescing (many-to-one) and splitting (one-to-many) requests cause the path of a request to include sub-paths of other requests. As discussed in Section 3.2 write requests are often coalesced. Re-creating the full path of an original write request currently re-creates the path of all other requests that were coalesced with it. Hence, the cost of re-creating the write path is usually larger than the read cost. Some workloads do not exhibit sequential behavior (e.g., Postmark) and little or no coalescing happens for them. In practice, we have seen path creation times ranging from a few microseconds (when the breadcrumbs are still in the buffer cache of the Activity DBs) to a few milliseconds.

Figure 9 shows the latency graph taken by NFS_Create calls from a specific client for the Postmark workload. Such paths can be reconstructed online by querying the ActivityDBs.

5.4 Trace pruning methods

The trace-based approach of measuring performance allows for easy integration into a system. However, the system is not expected to maintain all raw traces at all times, since they consume storage space. From our experience with using these traces, we have found that pruning the CPU, network, and disk traces to generate per-client performance resource and latency information is acceptable.

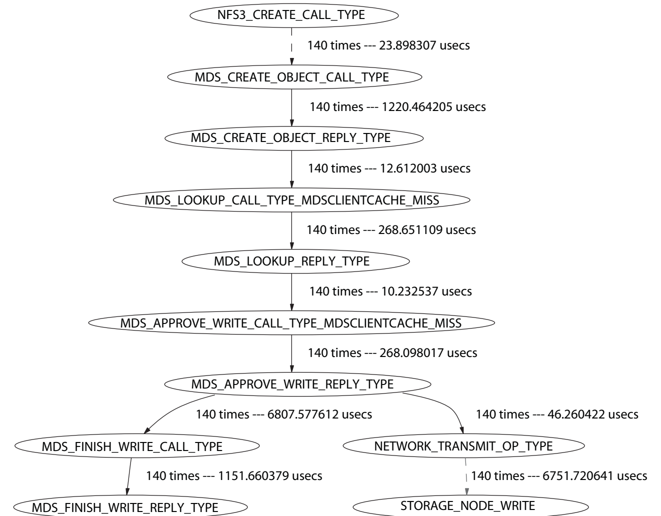


Figure 9: Example path created by NFS Create calls. This path was obtained using the Postmark workload and averaging 140 paths. The nodes contain information such as the unique identifier of the component posting the record and the string name of the record. The edges contain latency information between two instrumentation points. Some requests may be processed in parallel. A dashed line indicates that the request is moving to another physical machine. These graphs are automatically generated using the trace data and a readily available visualization tool like DOT [17].

However, we keep full buffer cache and disk traces as well.

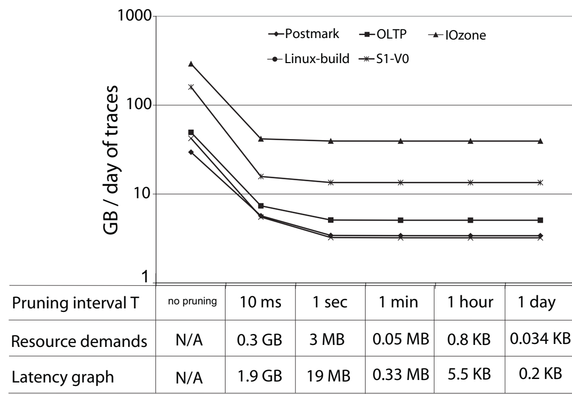
Figure 10 shows the storage demand when the traces derived from the workloads are pruned every T units of time. As T increases, the amount of trace data decreases proportionally. The graph shows the amount of trace data from keeping pruned CPU, network, and disk traces and also full buffer cache and disk traces (pruned disk traces reveal disk utilization metrics, whereas the full traces can be used for trace re-play). The table beneath the graph shows that further savings can be made if the full buffer cache and disk traces are not kept. The table shows just the cost of traces if only the per-client, per-resource demands are kept, or only the latency graphs are kept. Section 6 discusses cases when we have found it useful to keep the full buffer cache and disk traces, in addition to the resource and latency information.

6. EXPERIENCES

This section reports on our experiences and lessons learned with Stardust.

Predicting effects of workload and resource changes: In Ursa Minor, Stardust serves as the tracing and querying infrastructure for several automation agents. In particular, these agents are responsible for answering *What*...*if* questions about hypothetical workload and resource changes [42]. Concrete *What*...*if* questions we solve include:

- "*What* would be the performance of client A's workload *if* we add client B's workload onto the storage-nodes it is using?"
- "*What* would be the performance of client A's workload *if* its data is re-encoded from 2-way replication to a RAID-5 scheme?"
- "*What* would be the performance of client A's workload *if* the administrator buys more RAM/faster disks?"



Pruning interval T	no pruning	10 ms	1 sec	1 min	1 hour	1 day
Resource demands	N/A	0.3 GB	3 MB	0.05 MB	0.8 KB	0.034 KB
Latency graph	N/A	1.9 GB	19 MB	0.33 MB	5.5 KB	0.2 KB

Figure 10: Savings from pruning. The graph illustrates the storage savings from pruning the traces to get demand and latency information. Full traces of buffer cache and disk accesses are still kept, however. The table beneath shows the amount of data needed for just resource demands and latency information.

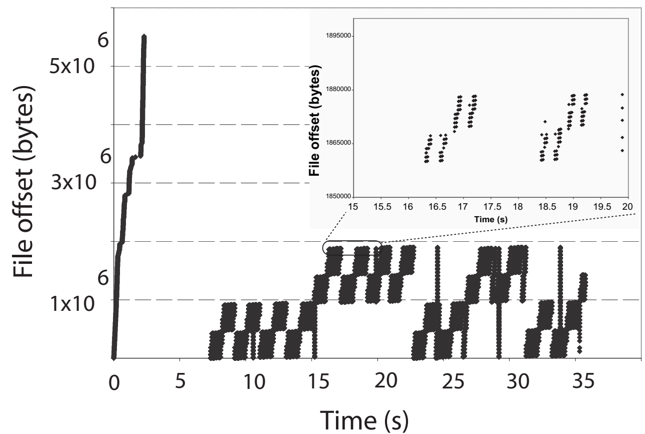


Figure 11: View of workload access patterns for the scientific workload. The workload shows strong spatial and temporal locality, hence client-side caching is the most effective optimization.

Answering such questions involves understanding per-client, per-resource demands, which the ATI collects and maintains. We found it necessary to have the full buffer cache and disk traces for these predictions. Those traces were needed to simulate the effect of hypothetical buffer cache and disk changes on performance using trace-replay techniques.

One realization from this work, is that, in a storage system like Ursa Minor, it is possible to predict how much CPU a client will consume from its requests' size and rate alone. CPU is consumed both at the client (for encoding the data as described in [42]) and at the storage-nodes (for checking data checksums to verify data is not corrupt). Our work showed that the CPU consumed at the client has a direct relationship with the data encoding chosen. At the storage-nodes, the CPU consumed per request has a direct relationship to the request's size and arrival rate. Hence, collecting detailed context switches may not so important for a storage system. Thus, the number of records and their storage size can be drastically reduced (as shown in Figure 6). However, in a different system, that information may still be useful.

Handling user complaints: We illustrate a type of complaint a storage administrator may get by examining a concrete complaint we have received. One of our users, running the scientific workload (S1 and V0 queries), complained about the performance of our system, hinting that the scientific workload ran faster on a local off-the-shelf file system than our cluster storage system. Such of comparison-based complaints are commonplace and a large time sink for administrators because of limited information from the system and the user.

Using Stardust, we first looked back in time to focus only on the requests of the client at the time of the complaint. Going back in time is straightforward, since all traces are kept and correlated, unlike the current approach of keeping logs and disconnected performance counters.

A query to get the maximum utilization of all system resources revealed that they were underutilized at all times (i.e., we looked at the full distribution, not just averages) and thus the problem was not the result of a bottleneck. The latency map of the requests revealed that the component-to-component latencies did not differ significantly from the average aggregate latencies observed on the system, hence the problem was not related to blocking.

Because all resources were underutilized and blocking was reasonable, the next step was to look at the client access patterns to understand the nature of the client requests. Figure 11 illustrates the locality of client requests through time. After the initial spike in the graph (showing file creation), all other accesses were small (48-92 bytes). This graph revealed the source of the problem. The client was using NFS version 3, which does not do client-side caching. Hence, these small requests, which are cached and coalesced into bigger requests in the local file system case, were sent to the cluster storage system directly, adding a significant number of network hops. We suggested that the user use NFS version 4 which would handle the client-side caching automatically when it becomes available (instead they implemented a simple application-level write-back cache).

Utilizing our instrumentation framework had several benefits over traditional methods. First, we did not need to bother the user with rerunning the workload for us to debug it in real time. Because our system keeps all traces, it was easy to go back in time. Second, we can extract the request latency map in a shared environment, which is not possible using the traditional methods. Third, we were able to generate the full distribution of access locations using a single SQL query. Performance counters and logs fall short of providing this flexibility. This was also a case where keeping just per-client, per-resource performance counters was not sufficient. We needed to look at the client NFS traces for root cause analysis.

Propagating breadcrumbs: Our first design involved propagating breadcrumbs through APIs to reconstruct the request path through different machines and across machines software modules. We discovered that the owners of the components were resistant to changing their APIs to add the breadcrumb structure. Our current approach does not involve propagating the breadcrumbs through APIs, but rather through the private data structures associated with each user-level thread. A request is usually processed by a single thread that moves it from one software module to the next. Throughout the processing, the breadcrumb of the request remains the same, hence we keep that inside the thread structure. Of course, when a request spans machines, the RPC layer still needs to propagate the breadcrumb to the new machine.

Another approach which was considered, but not implemented, was described by Isaacs et al. [22]. It consists of temporal joins on attributes that make up an activity record. That approach was subsequently used in the Magpie framework [7]. Such an approach does not require passing breadcrumbs around, making it more ele-

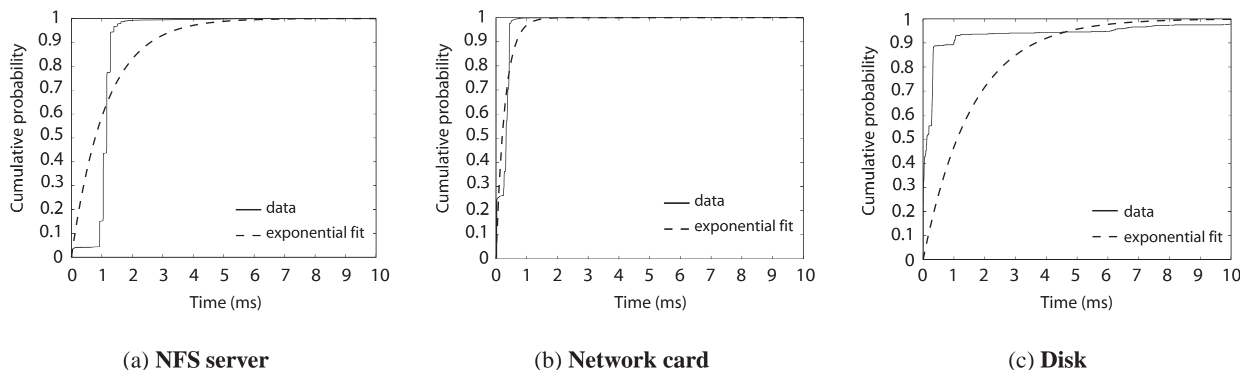


Figure 12: Request interarrivals at different points in the system.

gant in that respect. There are fundamental tradeoffs between Magpie's approach and Stardust's. Magpie's approach is more generic and flexible. The parser that stitches requests together in a chain can look at any arbitrary attribute when performing the join. Our approach is specific to a distributed storage system, where we know the types of requests the system will see. We trade off flexibility in describing types of requests with efficiency of querying. It is well-known in the database literature that joins are expensive operations; our approach queries based on breadcrumbs locally and only uses joins when a request moves to another machine. Magpie's approach requires joins to track a request even within a single machine. However, Stardust needs more information to be recorded (the breadcrumb) in each record.

Furthering future research: We believe that traces of activity taken at different points in a system are beneficial for furthering research in performance analysis in several ways. First, many tuning algorithms use trace-replay techniques, which require full traces as a way to evaluate a hypothetical system change [31, 34, 36, 37, 38]. Second, research in system performance, queueing models [24], and capacity planning [26] relies on real system measurements and traces. With most systems, these traces are taken at isolated points, for example at the disk level [19] or file system level [12, 45], but not at both simultaneously. Such measurements create a limited understanding on how the system operates as a whole and may lead to local solutions that do not work well when taken system-wide.

Figure 12, for example, shows measurements taken from the V0 query at three points in the system: the NFS server, where the request arrives; the network card, just after the I/O encode/decode operations happen; and the disk, inside the storage-node. A best-fit exponential distribution is also shown for each of the measurements. These measurements show that the exponential assumption does not always hold. When it holds, it may allow the use the $M/G/1$ queuing model to predict for response times [45]. However, rather than designing a predictive model for a particular arrival process, a system should be able to either predict when not to predict or choose the queueing model among many that best fits the actual workload. Stardust helps with keeping track of request arrival history.

What Stardust did not help with: Stardust does not help more than existing software profiling tools with finding or fixing algorithmic problems in the software. A poor implementation of the networking layer, for example, may mean that a request spends most of its time using the network resources. Stardust will identify the network as the source of latency, but will not make better suggestions on how to fix the problem than gprof in Linux [16] or Apple's Shark tool [6]. These tools already do a good job in informing software developers for paths that need to be optimized in the code that comprises a single local process.

7. SUMMARY

Stardust is an infrastructure for collection and querying of per-request end-to-end traces in distributed systems. Such traces better support performance diagnosis and tuning tasks by enabling generation of per-workload latency graphs and per-workload, per-resource demand information in addition to traditional usage counters. Experiments with diverse system workloads show that such fine-grained tracing can be made efficient. Experiences with using Stardust's traces for example administrative scenarios confirm their value for on- and off-line analysis of system behavior.

8. ACKNOWLEDGEMENTS

We thank James Hendricks for insights on the RPC scheduling for the ATI's traces. We thank the many supporting users of SQLite, and especially Dr. Richard Hipp, for helpful tips. We thank the anonymous reviewers for their useful feedback. We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate and Sun) for their interest, insights, feedback, and support. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389, by NSF grant number CNS-0326453, by DARPA's SRS program via Air Force contract number FA8750-04-01-0238, and by the Air Force Research Laboratory via contract F49620-01-1-0433. The data and traces obtained for the experiments in this paper can be obtaining by contacting the authors.

9. REFERENCES

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies, pages 59–72. USENIX Association, 2005.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. ACM Symposium on Operating System Principles, pages 74–89. ACM Press, 2003.

[3] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High Resolution Forward and Inverse Earthquake Modeling on Terascacale Computers. ACM International Conference on Supercomputing, 2003.

[4] N. Allen. Don't waste your storage dollars: what you need to know, March, 2001. Research note, Gartner Group.

[5] E. Anderson and D. Patterson. Extensible, scalable monitoring for clusters of computers. Systems Administration Conference, pages 9–16. USENIX Association, 1997.

[6] Apple. Optimize with Shark: Big payoff, small effort, February 2006. http://developer.apple.com/tools/.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. Symposium on Operating Systems Design and Implementation, 2004.

[8] J. P. Bouhana. UNIX Workload Characterization Using Process Accounting. 22nd International Computer Measurement Group Conference, pages 379–390, 1996.

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. USENIX Annual Technical Conference, pages 15–28. USENIX Association, 2004.

[10] M. Y. Chen, E. Kiciman, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. International Conference on Dependable Systems and Networks (DSN'02), pages 595–604, 2002.

[11] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: a building block for automated diagnosis and control. Symposium on Operating Systems Design and Implementation, pages 231–244. USENIX Association, 2004.

[12] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies, pages 203–216. USENIX Association, 2003.

[13] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[14] Gartner Group. Total Cost of Storage Ownership — A User-oriented Approach, February, 2000. Research note, Gartner Group.

[15] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. International Conference on Dependable Systems and Networks, 2004.

[16] Gprof - The GNU Profiler, November 1998. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html.

[17] GraphViz. Graphviz - Graph Visualization Software, February 2006. http://www.graphviz.org/.

[18] J. L. Hellerstein, M. M. Maccabee, W. N. Millsii, and J. J. Turek. ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems. International Conference on Distributed Computing Systems, pages 152–162. IEEE, 1999.

[19] HP Labs (Storage systems). Cello disk-level traces, 2005. http://www.hpl.hp.com/research/ssp/software.

[20] C. Hrischuk, J. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pages 399–409, 1995.

[21] IBM. DB2 Performance Expert, 2004. http://www-306.ibm.com/software.

[22] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. 11th ACM SIGOPS European Workshop, pages 92–97, 2004.

[23] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[24] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queuing network models*. Prentice Hall, 1984.

[25] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, **30**(7), July 2004.

[26] D. Menasce and V. Almeida. *Capacity planning for web performance: metrics, models and methods*. Prentice Hall, 1998.

[27] Microsoft. Event tracing, 2005. http://msdn.microsoft.com/.

[28] Microsoft. Microsoft .NET, 2005. http://www.microsoft.com/net/default.mspx.

[29] Microsoft. Windows Server 2003 Performance Counters Reference, 2005. http://www.microsoft.com/technet/.

[30] S. Microsystems. Java 2 Platform, Enterprise Edition (J2EE), 2005. http://java.sun.com/j2ee.

[31] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. ACM SIGCOMM Conference, pages 51–61, 1997.

[32] W. Norcott and D. Capps. IOzone filesystem benchmark program, 2002. http://www.iozone.org.

[33] Oracle. Oracle Database Manageability, 2004. http://www.oracle.com/technology/.

[34] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[35] Quest Software. Application Assurance, October, 2005. http://www.quest.com.

[36] B. Salmon, E. Thereska, C. A. N. Soules, and G. R. Ganger. A two-tiered software architecture for automated tuning of disk layouts. Algorithms and Architectures for Self-Managing Systems, pages 13–18. ACM, 2003.

[37] A. D. Samples. Mache: no-loss trace compaction. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **17**(1):89–97, May 1989.

[38] K. W. Shirriff and J. K. Ousterhout. A trace-driven analysis of name and attribute caching in a distributed system. Winter USENIX Technical Conference, pages 315–331, January 1992.

[39] SourceForge.net. State Threads Library for Internet Applications, February 2006. http://state-threads.sourceforge.net/.

[40] SQLite. SQLite, 2005. http://www.sqlite.org.

[41] Sun Microsystems. *NFS: network file system protocol specification*, RFC–1094, March 1989.

[42] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. International conference on autonomic computing, 2006.

[43] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis, published as Technical Report CMU-PDL-05-108. Carnegie Mellon University, October 2005.

[44] Xaffire Inc. Web session recording and analysis, October, 2005. http://www.xaffire.com.

[45] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. ACM Symposium on Operating System Principles, pages 177–190, 2005.