SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring

Sandip Agarwala and Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, schwan}@cc.gatech.edu

Abstract

Runtime monitoring is key to the effective management of enterprise and high performance applications. To deal with the complex behaviors of today's multi-tier applications running across shared platforms, such monitoring must meet three criteria: (1) fine granularity, including being able to track the resource usage of specific application behaviors like individual client-server interactions, (2) real-time response, referring to the monitoring system's ability to both capture and analyze currently needed monitoring information with the delays required for online management, and (3) enterprise-wide operation, which means that the monitoring information captured and analyzed must span across the entire software stack and set of machines involved in request generation, request forwarding, service provision, and return.

This paper presents the SysProf *system-level monitoring toolkit, which provides a flexible, low overhead framework for enterprise-wide monitoring. The toolkit permits the capture of monitoring information at different levels of granularity, ranging from tracking the system-level activities triggered by a single system call, to capturing the client-server interactions associated with certain request classes, to characterizing the server resources consumed by sets of clients or client behaviors. The paper demonstrates the efficacy of SysProf by using it to manage two different enterprise applications: (1) detecting performance bottlenecks in a high performance shared network file service, and (2) enforcing service level agreements in a multi-tier auctioning web site.*

1 Introduction

Distributed systems are becoming increasingly complex, in part because of the prevalent use of web services, multi-tier architectures, and grid computing [3, 13, 17], where dynamic sets of machines interact with each other via dynamically selected application components. A key problem in this domain is to understand the runtime performance behaviors of these highly distributed, networked applications and systems, to better manage system assets or application response and/or to reduce undesired effects like congestion.

Multiple technical issues make it difficult to manage enterprise applications. First, effective management often requires detailed performance analysis, going beyond measurements of simple metrics like average CPU load, network bandwidth, number of tasks completed, etc. [6]. Yet there currently exist no standard tools for capturing such detailed information. Second, while industry is developing standards for instrumenting distributed applications and systems [5], the XML-based representations used for the Common Base Event monitoring standard and the application-level monitoring methods and interfaces provided by widely available tools like HP Openview exhibit overheads that prevent their usage for capturing and analyzing detailed system- or application-level information about the precise resource usage associated with select application behaviors [24]. Third, the large diversity of applications routinely used in the enterprise and grid domains, ranging from simple *ftp* to complex distributed collaborations, makes it difficult to assume the existence of common, clean interfaces for performance evaluation. Finally, source code is not likely readily available for all of the applications being evaluated and managed by an organization.

One approach to runtime management is to integrate generic methods for analyzing program performance into middleware, used in systems like Photon [27], Pinpoint [12], and many others [28]. The idea is to automatically observe a program's usage of middleware functions, including the middleware-mediated interactions between different, distributed application components. Applications need not be modified, and access to source code is not necessary. However, since actual resource usage is controlled by the operating system, it is not possible to accurately account for the performance effects of certain application- or middleware-level behaviors. The basic causes of these problems are system-level asynchrony, i.e., the OS kernel's internal use of concurrency to satisfy multiple application requests, and system-level independence, i.e., the fact that OS kernels independently manage and allocate system resources for the multiple application-level processes being run. From the middleware level, therefore, it is difficult to attribute the usage of certain system resources to specific user-domain actions.

Prior research work has already recognized the importance of making operating systems more flexible and accountable for their resource usage [7, 21]. The goal of our research is to provide to applications accurate and timely information about their current resource usage. There are many uses for such information. A system administrator may be interested in the amount of time a client's request spends inside the OS kernel, to detect why a web-server is responding too slowly. More generally, shared network services in multi-tier architectures will concurrently execute requests from different clients, and request processing is not limited to just one machine. Finally, in addition to analyzing the performance implications of the complex distributed behaviors listed above, information about total resources used in processing requests is very important for utility billing, auditing, enforcing service level agreements (SLA), capacity planning and other management tasks.

The specific question asked in our work is whether it is possible to dynamically gather detailed monitoring information about shared network applications and then analyze their behavior, without having complete knowledge about their design and structure and without studying their source code (if available). That is, are there general ways to capture and analyze application behavior without having to instrument the application, instrument middleware, or make assumptions about application APIs? To answer these questions, we have designed the *SysProf* system-level toolkit, which provides a flexible framework for measuring the resource consumption behavior of various activities. An *activity* may be a *system call* made by some user-level application, or it may be a specific request-response interaction between a client and a web service. An activity may also be some class of application-level actions, such as the composite behavior of requests residing in a high priority request queue in an application server. In all such cases, SysProf provides support for carrying out enterprise-wide measurements – from application to system levels and across multiple machines – of the resources used by activities. SysProf's interface is such that activity monitoring may be customized, at runtime, to current needs. Furthermore, with the monitoring of runtime activities may be associated the analyses needed to aggregate, filter, or correlate monitoring data, as per current diagnostic needs. Analyses are carried out by pre-built kernel-level functions that can be dynamically activated or de-activated, and/or they can use custom functions specified by the application or system administrator. Furthermore, after local, in-kernel analysis, monitoring data may then be aggregated and sent to remote analyzers (or to any remote data consumer) through kernel-level publish-subscribe channels. These channels potentially connect all machines participating in the activities being carried out. In essence, therefore, SysProf uses a system-level overlay to capture, analyze, and correlate monitoring data. The overlay's actions may be dynamically customized to meet the granularity and real-time needs of the processes that require monitoring information.

SysProf does not require changes to user-level code, including changes that would recompile it (e.g., with a debug switch). By using system-level mechanisms for monitoring user-level applications, SysProf can run without user involvement and without source code knowledge. Another advantage of SysProf is its ability to collect richer and more accurate information than is possible at user level. This includes tracking in detail the actions of specific dynamically selected applications, application components, and properties of their behaviors.

SysProf is derived from our earlier work on kernel-level monitoring, termed DProc [1]. Compared to such work, the new contributions described in this paper are the following:

- SysProf provides a flexible framework for monitoring at the granularity of individual *activities*, such as the system calls issued by a specific client or a client's interactions with a certain remote application service.
- SysProf's analysis actions associated with the runtime capture of monitoring data are configurable dynamically, thereby enabling tradeoffs between the granularity, overheads, and delays of runtime diagnoses.
- High performance and low perturbation for low granularity monitoring are due to SysProf's use of dynamic code generation, binary encodings for monitoring data, low overhead kernel-level publish-subscribe messaging, and efficient event hashing.
- The utility of SysProf is demonstrated in two application contexts:(i) in a shared NFS service where SysProf can dynamically detect bottlenecks in proxies vs. servers, and (ii) in a multi-tier auctioning web service called RUBiS [10], where SysProf-based runtime monitoring and diagnosis are used to improve the scheduling of client requests.

Micro-benchmarks and performance evaluations of SysProf validate the importance of low granularity and highly accurate monitoring. The overhead of SysProf is within acceptable limits that makes it possible to be applied to many online algorithms. In our evaluation, application performance of an online E-Commerce website decreased by less than 2% because of SysProf. But the throughput gain ($> 14\%$) that was achieved with SysProf far outweighed the cost. SysProf was also able to determine the bottlenecks in a virtual storage service by correctly identifying the sources of latencies in the system.

2 SysProf: Design and Architecture

The **SysProf** toolkit keeps track of the different *activities* in a distributed system and resources consumed by them. An *activity* may involve just one machine, like a system call that reads file data from a local disk, or it may span multiple machines, like a "HTTP" request in a multi-tier web service. In either case, an activity is a SysProf-defined entity that is not constrained to match a single application-, middleware-, or system-level abstraction. This paper focuses on activities that involve network interaction between multiple machines. Our ongoing work is using the activity notion to better understand end-to-end application properties in the light of concurrent OS behavior on single machines.

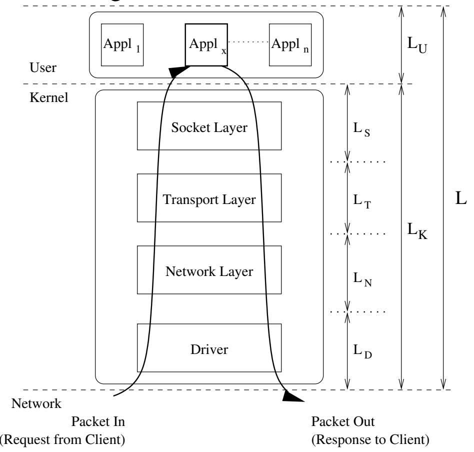

Figure 1: An Activity example: Different L's show the time spent (latency) at each of the marked steps

Figure 1 presents a sample activity. A request packet from a client arrives in a system and after being processed by different network protocol layers, it is delivered to the user-level server , Appl$_x$. The server performs some computation on the request and calculates a response, which is then sent back to the client, after again traversing the network stack. At each processing step, some resources (e.g. CPU cycles, memory, etc.) are used, and the request may be queued a number of times before a response is finally sent out. In order to debug the performance of this application and detect potential bottlenecks, the developer or the system administrator may need to know the time spent and resources consumed at each of these steps. In addition, the developer may need to understand queuing and concurrency behaviors. SysProf can provide details about the time spent in different steps of the network protocol processing, time spent by application at the user-level and at the kernel-level while the request is being processed, time spent by the application waiting for I/O during request processing, etc. SysProf monitoring requires no modification to the Appl$_x$ while providing detailed insights into the execution of the client's request. (1) Monitoring information can be used to identify bottleneck resources, by identifying where most of the time is spent (i.e., at the kernel-level or at the user-level). (2) It can identify the reason a request spends some unusual amount of time in the kernel buffer, perhaps because there are too many outstanding requests Appl$_x$ must process or perhaps because of some bug in the Appl$_x$ itself. (3) It can identify what Appl$_x$ was doing when the request was waiting in the kernel buffer? Was it executing or was it blocked for some reasons (e.g. I/O)? Answers to such questions are important steps toward identifying performance problems in networked IT infrastructures.

The depiction of the communication activity shown in Figure 1 is oversimplified. Actual web service requests, for example, may be processed locally by the server by fetching data from local disks, or they may query database servers on remote machines before responses are generated. Such requests may be processed asynchronously by processes different from the ones who originally received them (e.g., proxies), and control transfers may be accomplished by shared memory, message queues, or with other IPC mechanisms. Other issues like concurrency (to handle requests from different clients) and interleaving (handling different requests from same client) further complicate request analysis. Figure 2 depicts an overview of the SysProf architecture. It has five main components, which are described in detail below:

Kprof is the SysProf monitoring interface. It operates at kernel level and provides a generic API for the collection of various events from different kernel components. To track activities, a set of key points in the kernel are instrumented statically (like in Linux Trace Toolkit [30]). Kprof receives information from these points as efficient binary events. Events are delivered by invoking a function provided by Kprof's API. These events can be grouped into four major types: Scheduling events (context switches, process creation/deletion, etc.), System Call events, Network events, and File System events (open, close, read, write, etc). Events can be selectively switched *on* and *off* depending on the requirement set by the SysProf controller or the *local performance analyzer* (LPA). Events can also be pruned on the basis of process IDs, group IDs, or other such predicates. Each LPA specifies the set of events in which it is interested by registering a callback function with Kprof. These callbacks are invoked by Kprof when their events are generated. When none of the analyzer(s) subscribes to events, all of them are turned off, resulting in almost negligible perturbation for Kprof-instrumented operating system kernels. Kprof builds on our earlier dProc kernel-level monitor, and its functionality is similar to the static kernel instrumentation offered by LTT [30]. Further, using Kprof does not prevent us from using other available instrumentation techniques like Dprobes [22], Dtrace [9], Kerninst [26], etc. Our goal is not to innovate in kernel-level monitoring, but instead, to have sufficient facilities for extracting relevant monitoring information from OS kernel, without major kernel modifications and with acceptable perturbation [19].

The **Local Performance Analyzer** filters, aggregates, and

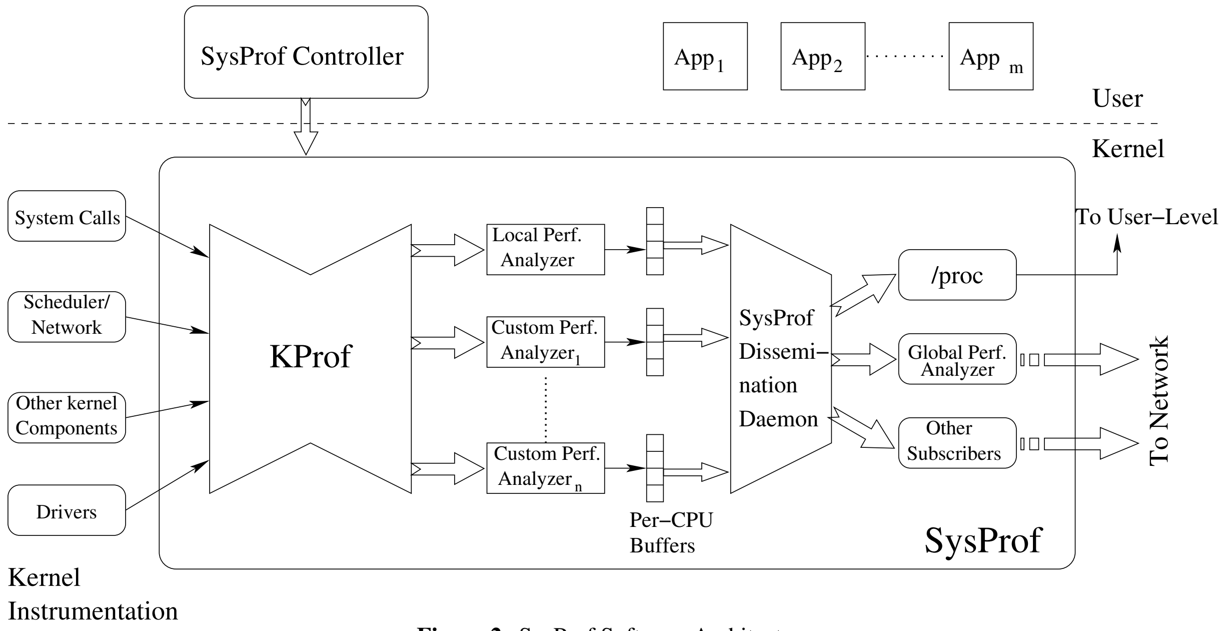

Figure 2: SysProf Software Architecture

correlates raw monitoring data, and then uses it to generate different performance metrics from the events generated by Kprof. There can be more than one LPA in SysProf, each potentially performing different analyses. During initialization, each LPA registers a callback with Kprof, and it specifies a list of events that need to be delivered to it. These callbacks are in the "fast path" of the kernel code and may also be invoked from interrupt contexts. Therefore, it is necessary that they never block and are computationally small. For lack of space, we next describe only one LPA in detail, the one that diagnoses a request-response interaction between a remote client and a user-level application server.

Messages and Interactions: The first diagnosis step is to identify a certain request-response pair. Because of interleaving and concurrency (as discussed in Section 2), it is non-trivial to extract such a pair without any application-specific knowledge. Recently, some offline black-box approaches have been proposed to infer causal path patterns [2], but online black-box techniques pose challenges like overhead and timeliness. In order to enable online analysis, SysProf defines the notions of *messages* and *interactions*. Let node_A (identified by $\{\text{node}_A$ IP, node_A port$\}$ pair) and node_B (identified by $\{\text{node}_B$ IP, node_B port$\}$ pair) be the nodes communicating with each other. A series of packets from node_A to node_B without any intervening packets in the opposite direction constitute one *message*. An *interaction* consists of a message pair in the opposite direction. The intuition behind this approach is that requests and responses will be composed of multiple packets. Multiple requests may interleave, in which case domain-specific knowledge and/or ARM support [5] would be necessary.

LPA subscribes to multiple Kprof events to keep track of *interactions* and the different performance metrics associated with them. Specifically, LPA maintains a window containing the past several interactions and the metric values computed for them. Window size can be changed dynamically, and window contents are evicted to the dissemination daemon after some time. That is, each LPA maintains two per-CPU buffers to store captured data, and when one of them has been filled, the dissemination daemon is notified, and the LPA switches to the next buffer. Each such buffer switch requires interrupts to be disabled locally to avoid data corruption.

Information collected from these events includes the time at which some interaction started, the number of packets/bytes exchanged, the amount of time spent by the interaction in user and kernel modes, the interaction id, the name and the function of the user-level application server that receives packets from the interaction, and others. It is also possible to capture information about context switch details, the number of disk I/O operations performed by the application, and the length of time the application is blocked (e.g., for I/O) during an interaction. Such information capture can be configured and turned on and off dynamically, depending on current analysis requirements.

Custom Local Performance Analyzer (CPA): In addition to the statically defined LPAs, custom analyzers can be dynamically created and downloaded into the kernel. CPAs function just like normal LPAs, including registering of callbacks with Kprof and indicating the set of events they wish to receive. CPAs are specified in the form of E-Code [14] (a language subset of C), compiled through run-time code generation. CPAs provide great flexibility in terms of specifying

application-specific analyses.
The **SysProf dissemination daemon** distributes the information generated by LPAs to the remote nodes that need it and also makes it available to the user-level through the standard "/proc" virtual filesystem interface (i.e., as with Dproc [1]). On receiving a "*buffer full*" notification from a LPA, the daemon wakes up and copies the LPA's data into its own buffer. If the data is not picked up in a timely fashion, it may be overwritten. The size of the buffer, therefore, must be chosen carefully. Remote nodes subscribe to the information generated by LPAs, and it is the daemon's job to aggregate data collected from different LPA buffers in order to send it to interested parties. For high performance and low overheads in event acquisition and dissemination, the daemon uses dynamic data filters, PBIO-based binary encodings, and kernel-level publish-subscribe channels.
The **Global Performance Analyzer** aggregates and correlates the data it receives from different SysProf daemons. Specifically, it correlates the source and destination IP addresses, port information, and NTP timestamps in the logs from different nodes. After aggregating the resource usage of each individual interaction, GPA computes the overall performance of the associated request-response pair. Other nodes in the system can query the GPA to determine information about a particular interaction or about the system as a whole. The GPA periodically dumps its information onto local disk, which can be used later for purposes of auditing, workload prediction, and system modeling.
The **SysProf controller** regulates the granularity and the amounts of information monitored and analyzed by SysProf. It can instruct the LPAs to collect statistics for some client class rather than for individual interactions. It can change the sizes of internal LPA buffers. It provides a management interface for SysProf and makes it easy for the user to select its functionalities.

3. Experimental Evaluation

SysProf has been implemented in Linux (kernel version 2.4.19) as a set of loadable modules and a kernel patch that defines the instrumentation required to generate events. The current implementation only supports Intel x86 platforms, but the general technique is applicable to other architectures. The next few subsections describe the results of micro-benchmarks that assess the accuracy and overheads of SysProf. We then present our experiences with SysProf in detecting bottlenecks in a shared NFS application and in making scheduling decisions in an online auctioning web-site called RUBiS [10].

3.1 Microbenchmarks

As discussed in the previous sections, SysProf has a negligible effect on the performance of services it monitors. Because of its configurable interface, the overhead of SysProf can be varied ranging from less than 1% of the system resource to more than 10%. We measured the overhead in its default configuration by running it with *linpack* benchmark on a setup of two nodes (2.8GHz uni-processor, 512KB cache and 4GB RAM) connected to each other by a 1Gbps ethernet. *Linpack* measures the computation power of a machine in *MFLOPS*. There was no change in the *mflops* measured by linpack due to SysProf. One of the reasons is that SysProf generates more activities when there are network interactions, so linpack was probably not a very good benchmark. In another microbenchmark experiment, we employed *Iperf* to test the overhead of SysProf. Bandwidth was measured between two nodes, first with SysProf disabled and later enabling it. The measured bandwidth in the later case ($\sim$810 Mbps) was almost 13% less than that of the former ($\sim$930 Mbps). This reduction in bandwidth was due to overhead incurred by examining packets at such high speed and not due to SysProf network usage. In a 100Mbps LAN, this overhead came down to 3%.

3.2 Shared NFS Proxy

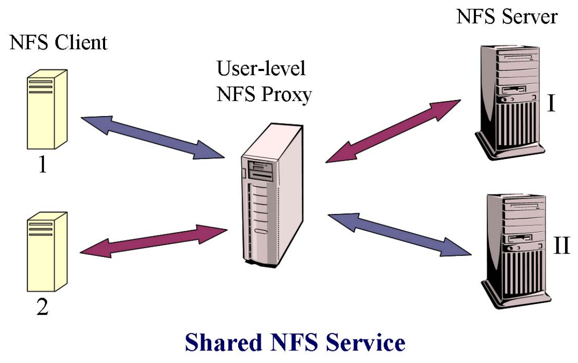

Figure 3: Virtual Storage Service

Virtual storage architecture has been quite popular in the enterprise storage domain to provide fast, efficient and fault tolerant service to the consumers [4]. Figure 3 shows a very simplified version of a virtual storage service. The back-end storage servers are hidden from the client's view by a user-level proxy that interposes every request from the client to the server. The real scenario is of course very complex and may consist of multiple level of hierarchy, each providing some kind of service and with multiple front-end proxies. A typical problem in these environments is to detect failures and performance bottlenecks. One of the ways to detect this is by tracking the execution of the requests through different components and measuring the latencies and resources consumed. However, this is a difficult problem because there may be a large number of nodes involved through which the requests pass and get processed.

In this section, we illustrate how SysProf can be used to detect performance bottleneck in a virtual storage service like the one shown in Figure 3. We ran a filesystem benchmark called *Iozone*(`http://www.iozone.org/`) as client's workload. *Iozone* benchmark can test the performance of the number of I/O operations. We configured *Iozone* to generate write/re-write tests and varied the number of threads it forks to see the effect on resource usage. The number of threads created in each runs were same for both the clients.

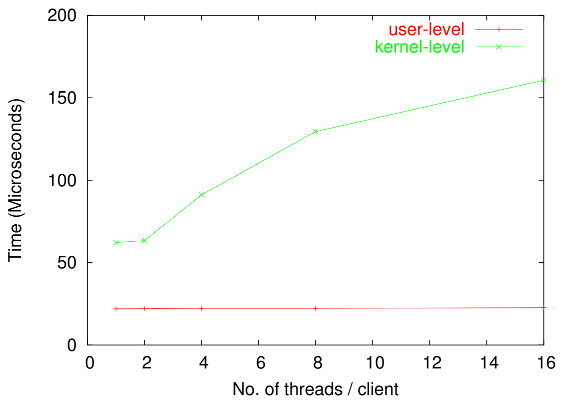

Figure 4: Avg time spent by client-proxy interactions at the proxy

Figure 4 shows the average amount of time an *interaction* between client and proxy spend at the proxy node, both at the user-level and at the kernel-level. The amount of time a request spent at the user-level is almost constant for different number of client threads but the kernel time goes up because of increase in the request traffic. This is because the proxy does very little processing of requests and its job is to just forward the request to the back-end NFS servers. Therefore, it spends a constant amount of time on every request it processes. But as the traffic is increased, kernel buffers get filled up and the requests get queued at the kernel-level waiting for their turn to get processed by the user-level proxy.

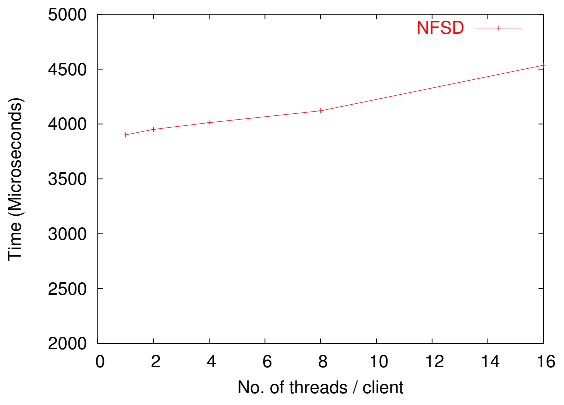

Figure 5: Avg time spent by the interactions at back-end server

We did a similar measurement with SysProf at one of the back-end NFS servers. Since the NFS server ran as kernel daemon, no time was spent by the request at the user level. Figure 5 shows the average time spent by different *interactions* in the kernel of the back-end server. This time is more than an order magnitude than the time spent in the proxy. This shows that the the back-end server is the major contributor to the delay seen in processing the client request. The network round-trip delay is insignificant ($< .3$ms) as compared to the time spent in the back-end.

From the above study, we showed how SysProf can be used to identify the bottleneck resources. It not only tells the delay incurred in request processing on a particular node but also gives fine details like whether the amount of time was spent in user-level or kernel-level, the number of outstanding interactions and so on.

3.3 Multi-tier Web Service

In this section, we study a dynamic window-constrained scheduling algorithm for a multi-tier web application called RUBiS, and show how it can provide better QoS using the information provided by SysProf. RUBiS implements core functionalities of an auction site like selling, browsing and bidding. RUBiS is available in three different flavors: PHP, Java HTTP Servlets and EJB. We use the Servlets version.

In this evaluation study, we apply a black-box scheduling algorithm called *DWCS* to RUBiS and demonstrate that a *resource-aware DWCS* can provide better QoS guarantees as compared to the ordinary *DWCS*. Although DWCS has traditionally been used in streaming multimedia applications that can often tolerate infrequent losses or misses of data generation or transmission and in linux process scheduling [29], it is equally applicable in enterprise domain where different workloads need to be multiplexed in a shared utility infrastructure (like a multi-tier web service). These workloads are often associated with some performance goals (like the minimum throughput or the maximum response time) and may have certain real-time requirements which are usually expressed in the form of Service Level Agreements (SLAs). For example, a *bidding* request in an online auction site like RUBiS has real-time deadlines, while a *comment* posted by a user has a less stringent deadlines.

We apply DWCS to schedule two different request classes in RUBiS with SysProf disabled. These requests were generated using *httperf* [23] on a separate client machine. 60 client sessions were created and half of them generated high priority *bidding* requests and the other half generated low priority *comment* requests. The *bidding* request is cpu intensive and consumes lot of cpu at the servlet server which processes it. The *comment* request on the other hand generates significant network traffic. Each request class has a *Poisson* arrival distribution with mean rate equal to 150 requests/sec. The scheduler ran on the same node as the client and the request dispatching was facilitated by prefixing the request's URL path with the appropriate servlet server's name. Apache server was configured to multiplex the requests to the different backend server depending on these prefixes. Figure 6 shows the result of applying DWCS to the two request classes. The average throughput achieved for *bidding* requests and *comment* requests were 145 and 134 responses/sec. respectively. Halfway through the ex-

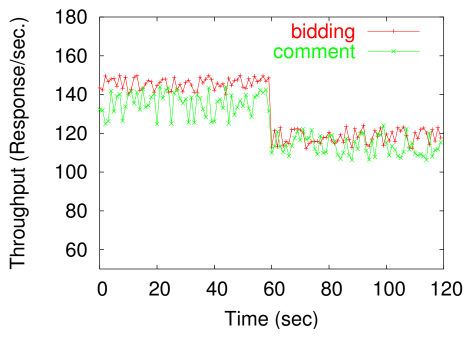

Figure 6: Throughput with DWCS

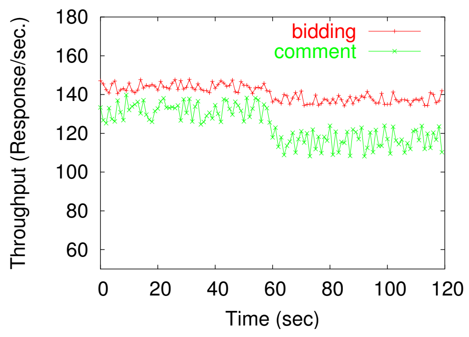

Figure 7: Throughput with RA-DWCS

periment, we introduce perturbation in one of the servlet servers by running four linpack(`http://www.netlib.org/linpack/`) processes. The average throughput of the two classes fell to 118 and 115 responses/sec. respectively.

Figure 7 shows the results of resource-aware DWCS (RA-DWCS) that use SysProf information to guide its scheduling and dispatching decisions. The degradation in throughput is far less as compared to our earlier experiment. It should also be noted that the higher priority *bidding* request has very insignificant drop in performance and this was basically because of the fact that these requests were routed by RA-DWCS to the server that was lightly loaded.

4 Related work

Performance monitoring of distributed systems is a frequent topic of investigation. The systems most similar to ours in terms of monitoring are Dproc [1], ganglia [20], MAGNeT [16] and some others. The differences between those systems and SysProf is that we can monitor and track resource usage both at multiple granularities and across multiple machines, and then analyze the resulting information in a hierarchical manner. The outcome is a low overhead monitoring solution.

The notion of request-based analysis is not a new one. It has been used in a number of research projects. Magpie [8] derives the causal paths and resource consumption from application, middleware, and system traces. Pinpoint [12] instruments J2EE middleware to propagate a unique id with each request, and then uses the generated traces to localize faults. In comparison, SysProf does not modify user-level code or instrument data packets. By doing so, we lose some causality information, but the resulting, low overheads allow us to perform online information analysis. Aguilera et al. [2] treat each system as a black box and infer the causal pattern from the passive message traces. However, this approach cannot attribute resource usage correctly because of the absence of internal system information.

Causeway [11] and SDI (Stateful Distributed Interposition) [25] provide operating system constructs that the application can use to track its activities in a multi-tiered system. The advantage is that it is possible to do very deterministic analysis with this approach. SysProf, on the other hand, tries to infer the application behaviour automatically and generates information at different level of granularity.

Many kernel instrumentation techniques have been proposed in the literature. Though the focus of our work is not in designing new instrumentation methods, they are still crucial to the basic performance of SysProf. SysProf uses LTT-like [30] methods of generating events. Finally, there are other tools like Dprobes [22], Dtrace [9], and Kerninst [26] that allow dynamic instrumentation of kernel code. Dynamic instrumentation may be desirable in cases where the system has to be debugged, but can't be shut down to apply static instrumentation patches.

Tipme [15] monitors and diagnoses unusually long latencies in an interactive environment on a single machine. ETE [18] requires application level instrumentation to generate end-to-end response times. In comparison, SysProf does not require changes to user-level code, and it can measure latencies and resource consumption on multiple machines. It analyzes performance data in a hierarchical fashion and provides a customizable interface that can be tuned at run-time.

5 Conclusions and Future Work

We presented a toolkit called SysProf that can monitor and analyze different activities in a distributed system at a different level of granularity. The kernel is instrumented to generate performance events that are processed, first by the local analyzers (in their per-CPU buffers) and then by the global analyzers. The toolkit is configurable and permit run-time extensions to add new analysis. The use of performance gears like the selective monitoring, hierarchical analysis, per-CPU buffers, kernel-level messaging and others keep the overhead low. However, certain activities (like the interleaved request) cannot be monitored efficiently without domain-specific knowledge. The toolkit was demonstrated to be useful in detecting performance bottleneck in a shared NFS service and in providing real-time guarantees in an enterprise-based web service.

Management of complex applications and IT infrastructures is becoming a key issue in the enterprise domain. Being able to automate the system and reduce human intervention can increase efficiency, reduce errors and significantly cut down IT costs. The next generation enterprise applications will be evaluated more on the basis of the ability to achieve QoS goals, Service Level Agreements and business revenue generated than on overall raw performance. This requires the system to be able to constantly monitor and analyze the services that are offered to the clients and give feedback to them, thereby forming a closed-loop system that can constantly adapt and tune itself to the changing workload, resources and business demand. The cumulative benefits in terms of decreased management complexity and higher quality of service easily offsets the cost due to monitoring overhead and with the recent trends in the hardware towards multi-core platform, it won't be unusual to have a core dedicated to the analysis of the services that run on that platform.

In the future, we want to continue with our research in formulating new algorithms to analyze distributed application behaviour. Our experience in designing SysProf has been valuable in identifying the challenges in this domain. We want to use this experience in building tools that can automatically discover and diagnose the problems in enterprise applications and make them more autonomic and manageable.

References

[1] S. Agarwala et al. Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms. In *HPDC*, pages 250 – 259, June 2003.

[2] M. K. Aguilera et al. Performance debugging for distributed systems of black boxes. In *SOSP*, October 2003.

[3] G. Alonso et al. *Web Services Concepts, Architectures and Applications*. Springer Verlag, 2004.

[4] D. C. Anderson et al. Interposed request routing for scalable network storage. *ACM TOCS*, 20(1):25–48, February 2002.

[5] Systems Management: Application Response Measurement (ARM). Open-Group Technical Standard, Catalog No. C807, ISBN 1-85912-211-6, July 1998.

[6] An architectural blueprint for autonomic computing, April 2003. `http://www-03.ibm.com/autonomic/blueprint.shtml`.

[7] G. Banga et al. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, pages 45–58, February 1999.

[8] P. T. Barham et al. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[9] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conf.*, pages 15–28, June 2004.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *OOPSLA*, Nov 2002.

[11] A. Chanda et al. Causeway: Support for Controlling and Analyzing the Execution of Web-Accessible Applications. In *Middleware*, November 2005.

[12] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, pages 595 – 604, June 2002.

[13] W. W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems 10, 1*, 3(20), January 1995.

[14] G. Eisenhauer. Dynamic Code Generation with the E-Code Language. Technical Report GIT-CC-02-42, Georgia Institute of Technology, College of Computing, July 2002.

[15] Y. Endo and M. I. Seltzer. Improving interactive performance using TIPME. In *SIGMETRICS*, June 2000.

[16] W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz. MAGNeT: A Tool for Debugging, Analysis and Reflection in Computing Systems. In *CCGrid*, pages 310–317, May 2003.

[17] I. Foster and C. Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[18] J. L. Hellerstein et al. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *ICDCS*, June 1999.

[19] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance Measurement Intrusion and Perturbation AnalysiPerformance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[20] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[21] J. C. Mogul. Operating Systems Should Support Business Change. In *HotOS X*, July 2005.

[22] R. J. Moore. A Universal Dynamic Trace for Linux and Other Operating Systems. In *USENIX Annual Technical: Freenix Track*, pages 297–308, June 2001.

[23] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[24] HP OpenView Transaction Analyzer performance and scalability Guide. `http://www.managementsoftware.hp.com/products/tran/`.

[25] J. Reumann and K. G. Shin. Stateful distributed interposition. *ACM TOCS*, 22(1):1–48, February 2004.

[26] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI*, pages 117–130, February 1999.

[27] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *SIGMETRICS*, pages 240 – 250, June 2002.

[28] IT responsiveness & efficiency with IBM WebSphere Extended Deployment, 2004. `ftp://ftp.software.ibm.com/software/webserver/appserv/library/WS_XD_G22%4-9126-00_WP_Final.pdf`.

[29] R. West, I. Ganev, and K. Schwan. Window-Constrained Process Scheduling for Linux Systems. In *3rd Real-Time Linux Workshop*, November 2001.

[30] K. Yaghmour and M. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *USENIX Annual Technical Conference*, June 2000.