sAccount: Resource Accounting of Shared Infrastructure in Multi-Tenant IT Platforms

Byung Chul Tak  Youngjin Kwon  Bhuvan Urgaonkar

Computer Systems Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802
{tak, yuk152, bhuvan}@cse.psu.edu

Technical Report CSE-11-013
November 2011

Abstract

In today’s consolidated IT platforms, the capability to accurately account the overall hardware resource usage among hosted applications can be necessary for a variety of resource management actions, and possibly also for auditing and explicit or implicit billing. We find that the increasing use of shared services in IT platforms, as exemplified by software-as-a-service offered by many clouds, renders existing solutions for accounting inadequate. We develop a resource accounting solution for a platform that consolidates multiple user applications and shared services on its server machines. We implement sAccount, a concrete realization of our solution, that consists of: (i) local monitoring facilities within the Xen hypervisor running on individual servers and (ii) a collective inference component that employs monitored data to derive accounting information for CPU and IO bandwidth resources. Using extensive evaluation on our prototype sAccount cluster involving a mix of synthetic and realistic (MySQL and HBase) shared services, we demonstrate the accuracy of sAccount. Our evaluation shows us that sAccount consistently offers less than 1% error in accounting at a low-overhead, whereas the error of a baseline technique employing state-of-the-art monitoring tools fluctuates between 5-150%. Furthermore, we also show that sAccount can enable administrators to improve the efficacy and agility of certain resource management tasks. For instance, sAccount allows correct and agile detection of the cause behind a CPU overload on servers running a shared MySQL database service. This, in turn, allows a simple resource control technique to correctly suppress the resource allocation for the source of this overload, whereas information offered by the baseline accounting leads to incorrect throttling decisions.

1 Introduction

An increasing number of information technology (IT) platforms consolidate multiple software applications on a shared set of hardware equipment for reasons of cost-efficacy or organizational necessity. Such consolidation and sharing occur in a wide variety of platforms such as (i) public clouds that “rent out” portions of their resources to their clients’ applications, (ii) private clouds that cater to the IT needs of multiple groups/departments internal to organizations, and even (iii) medium/small-scale data centers or clusters in enterprises or labs running multiple applications. While such sharing has always applied to shared networking and storage infrastructure - IO busses, network interface cards, storage-area-networks (SANs) - increasingly servers have also come to be shared heavily.

While consolidation offers numerous benefits [28], the resulting multiplexing of resource needs of different software entities - variously labeled crosstalk [19] or interference [13] - renders resource management more difficult. In particular, it becomes difficult to carry out accurate accounting of the resource usage of IT equipment,
Resource Accounting Information is Needed: Resource accounting information is needed for correctly charging the platform users’ applications - the chargeable entities (CEs) - for their usage. Such charging can be crucial (even necessary) for a variety of performance management, cost optimization, and anomaly identification tasks. It may also be needed in certain platforms for billing the chargeable entities based on their usage, either implicitly (e.g., as in an enterprise’s data center that is shared by applications catering to its internal groups/departments) or perhaps even explicitly (e.g., as in a public cloud that may wish to bill its clients for their applications’ usage of its resources). We formulate the resource accounting problem as a way to address this difficulty in shared IT platforms. Although we study the resource accounting problem in the context of shared servers, our work also provides insights for future work on accounting other shared hardware.

Figure 1: A portion of a platform that hosts two applications, each a CE, and the servers hosting their components. Arrows indicate communication between components. Also shown is a shared service - a database used by both the CEs. The shared service itself consists of multiple software components, some of which are exercised by the CEs indirectly (e.g., the “Data Store”), i.e., via requests made to other components (e.g., the “Front-end”).

where by accounting we mean ascertaining what portions of the overall usage of the platform’s hardware resources stem from (and should be attributed to) which applications.

Resource Accounting is Difficult: It is well-known that resource accounting is non-trivial even within a single server that consolidates multiple applications [4]. Often the work emerging from different user-level software entities is multiplexed in complex ways. It is often unclear how “charge” different user-level applications for the resources used on their behalf by other software including systems software (e.g., operating system or virtual machine monitors), runtime (e.g., garbage collector), or other shared services (e.g., shared file system, shared database server). The work on resource containers addressed this problem by modifying a server operating system to (i) allow applications to specify which software entities comprised a CE (a container), and (ii) implement monitoring functionality to record resource usage on behalf of such containers [4].

The resource accounting problem becomes significantly more formidable within a system consisting of multiple servers which run distributed applications, and where each server may host multiple software components. Since the resource usage for a CE possibly spans multiple servers, we must now (i) identify all such servers, (ii) carry out accounting locally at each of these servers, and (iii) correctly combine their locally-gathered accounting information per CE. A key difficulty in doing this arises because, in addition to application-owned software, platforms invariably also run a variety of (what we call) shared services - software components that are run by the platform to provide certain functionality to user applications or to the platform itself. Figure 1 shows a shared
database-as-a-service software, similar to what many current cloud platforms provide (e.g., SQL azure service), that is used by both CE_A and CE_B. These services are not owned by a particular CE but rather consume server resources on their behalf. A number of such shared services are offered by current cloud providers, particularly via the “software-as-a-service” (SaaS) model.

The presence of shared services renders all of (i)-(iii) above difficult. First, unlike application-owned software, a shared service may only be exercised by a CE indirectly, making it more difficult to ascertain this relationship. For example, in Figure 1, the front-end component of the shared database is invoked directly by both the CEs. Existing work, including ours [27], can be easily leveraged for identifying this relationship (if it is not already well-known for some reason). On the other hand, the data store is exercised via requests made to the front-end and not directly by the CEs. One approach for inferring such relationship may be to instrument the messaging S/W to inject tracking identifiers, which is not generally applicable and may be prohibitive.

Second, application-owned S/W components are contained within resource principals that are likely to be easily identifiable by underlying resource management software (e.g., the virtual machine monitors (VMMs) in Figure 1). This implies an existing local accounting solution such as resource containers can be easily used by this management software to associate these resource principals with the corresponding CE. E.g., in Figure 1, the web/app servers of CE_A and CE_B are hosted within their own virtual machines (VMs) created by the underlying VMM software; by associating containers with these VMs, existing accounting functionality can be leveraged. On the other hand, a shared service’s software design and configuration may not be amenable to easy adaption of existing solutions for local accounting. For example, the data store component highlighted in Figure 1 multiplexes the resources assigned to its internal schedulable entities (e.g., threads) in highly application-specific (and possibly unknown) ways among the activities it carries out on behalf of CE_A and CE_B, rendering a solution such as resource containers difficult to adapt.

Research Contributions: We develop an accounting solution that addresses all the difficulties identified above. One central concern in our design is to achieve generality by avoiding any application modification. We make the following key contributions.

1) We formulate the problem of resource accounting for shared servers running distributed applications. In particular, we incorporate the resource consumption of shared services into our problem definition. We introduce two novel and generic abstractions - the set of used servers for each CE, and the resource usage tree for each resource - that serve as useful vehicles for representing information needed by an accounting technique.

2) We propose the design of a distributed accounting infrastructure spanning the servers in the given IT platform. To the best of our knowledge, our design goes beyond the state-of-the-art by being the first accounting solution that is both: (i) completely implemented within privileged code (VMM in our case) with no need for application modification, and (ii) capable of accounting the usage of shared services. We implement sAccount, an accounting infrastructure based on our ideas within a cluster of servers, each running the Xen VMM and hosting virtual machines running Linux.

3) Using a mix of synthetic and real-world shared services, we show the efficacy of sAccount.

Outline: The rest of this paper is organized as follows. In Section 2, we define our resource accounting problem. In Section 3, we identify key design requirements. In Section 4, we present our design and implementation. In Section 5, we present our experimental evaluation. Finally, in Section 7, we present concluding remarks.

1In fact, this is the essential idea behind distributed resource containers [29]: individual servers use resource containers for local accounting and the network stacks within server operating systems are modified to embed tokens within messages sent to/by components that uniquely identify their CEs.
Figure 2: Examples of our accounting abstractions of “set of used servers” for a CE, and the “resource accounting tree” for each resource of interest within a server.

2 The Resource Accounting Problem

Terminology: Figure 2 shows the key entities and abstractions of interest to our problem. We show an illustrative portion of a public cloud (a representative multi-tenant IT platform) that hosts software applications for its clients. As mentioned earlier, we refer to a platform user’s application whose resource usage must be separately tracked and accounted as a chargeable entity (CE). Figure 2 shows two such CEs (labeled CE_A and CE_B) each a multi-tier e-commerce site. Each of these CEs supplies the cloud provider with a set of software components that the cloud runs on its behalf. Our platform runs each of these components within a VM, and this set of VMs is accommodated within physical servers s_1, ..., s_5. Each of these servers runs a VMM/hypervisor layer that multiplexes its server resources among overlying VMs.

Also shown in the figure is a shared service - a SaaS database - that the platform offers to its CEs. This shared service itself has multiple components that span servers s_6, ..., s_8. Although this database service is not a CE itself, we require our accounting solution to keep track of the resources it consumes on behalf of the CEs. Unlike for servers s_1, ..., s_5, where an accounting-capable VMM (e.g., using an existing accounting solution such as resource containers, as we described in Section 1) could associate the virtual machines V_1, V_2, and V_3 with CE_A and V_4, V_5 with CE_B, existing solutions cannot be directly adapted for accounting within servers s_6, ..., s_8 where the VMM-visible resource principals do not have a fixed association with any CE. Additionally, since the back-end tier of the database service is only exercised by the CEs indirectly, i.e., via work generated during processing of requests that are made by CEs to the front-end, additional thought is needed to identify what portion of its resource usage should be attributed to which CE.

Our Accounting Abstractions: We employ the following two abstractions to capture the distributed information that an accounting solution must keep track of:

1) Set of Used Servers: For each CE c, the accounting solution must maintains \( S_c(t) \), the set of servers whose resources are used for c over the time duration \([t, t + \Delta] \). As described already, this usage may be either: (i) direct, i.e., by one or more components of c, or (ii) indirect, i.e., by a shared service on behalf of c.

2) Resource Accounting Tree: For resource r within a server s, the accounting solution must maintain resource accounting information during the interval \([t, t + \Delta] \) in the form of a resource accounting tree \( T_{s,r}(t) \). We use the example shown in Figure 2 to explain a resource accounting tree for the CPU resource on server S_7. As shown,
the entire usage of the CPU resource within $S_7$ is represented by the root of the tree. The next level of nodes in the tree represent a breakdown of this overall CPU usage among three entities: (i) the VMM software, (ii) CE$_A$, and (iii) CE$_B$. The usage of VMM may be further broken down into portions attributed to applications CE$_A$ and CE$_B$ as captured by the nodes of the tree below the node for the VMM’s CPU usage. We denote the sum of the usage corresponding to all leaf nodes associated with the CE $c$ as $u'_c(t)$.

The aggregate resource accounting information for a CE $c$ during the interval $[t, t + \Delta]$ is simply the union: $U_c(t) = \bigcup_s \bigcup_{s \in R(s)} u'_c(t)$. The set of used servers for a CE as well as the resource accounting trees for an server instance may change with time, and sAccount must keep track of such changes. Changes to a resource accounting tree may be either structural (e.g., a VM migrates into the physical host) and/or in the values associated with various nodes in the tree (e.g., one CE starts to consume more CPU).

**Problem Definition:** Given a set of CEs and an accounting granularity $\Delta$, the goal of sAccount is to infer, for each CE $c$, the time-series $U_c(t)$.

### 3 Overview of Design Choices

Any accounting solution must have two elements: (i) local monitoring and (ii) collective inference. We use the phrase “local monitoring” to refer to facilities within each server that record events and statistics pertaining to the resource usage of (or on behalf of) each CE. E.g., in resource containers, local monitoring is carried out by the server operating system that is modified to identify resource allocation/scheduling events (e.g., when threads are scheduled/descheduled on the CPU) and using this information to charge their usage to appropriate containers [4]. We use the phrase “collective inference” to refer to functionality that is needed to combine the pieces of information offered by local monitoring to create a correct overall picture of accounting. Since resource containers are only concerned with a single server, collective inference is trivially realized from the monitored data. Distributed resource containers must address a more complicated version of collective inference, and it does this by augmenting the locally monitored data within each server with the identity of the distributed container (carried within messages exchanged between container components) that they correspond to [29].

As argued in Section 2, both local monitoring and collective inference need to be reconsidered for servers running shared services. There exist a large number of techniques and tools for local monitoring that one could choose from. In particular, these existing techniques span a wide spectrum of the “level of detail” they offer at the cost of generality, application intrusiveness, and overheads posed. At one end of this spectrum are techniques that can instrument user-space and OS/VMM code to create a very detailed record of a shared service’s resource usage that contains sufficient information for collective inference [5]. At the other end of the spectrum are CE-oblivious resource usage reporting tools that rely on information available within the server’s OS and VMM. E.g., top, and iostat. As we will empirically show in Section 5, collective inference that relies on data offered by these tools can have significant inaccuracies in accounting. Furthermore, as we will find, such inference can be extremely sensitive to a variety of system properties and environmental conditions, an undesirable feature. Although our results will be based on a specific inference technique, we argue that the root cause of these inaccuracies is the inadequacy of information contained in the monitoring information offered by these tools, and even more sophisticated inference techniques relying on such information would falter.

Generally speaking, collective inference is a statistical learning problem that must derive models that can meaningfully tie together the data provided by local monitors, possibly filling in any gaps or discrepancies within these data. The efficacy of such inference crucially depends upon the resource usage phenomena collected by local monitoring elements. Existing monitoring tools that are not application-intrusive have been designed for information collection at the granularity of OS/VMM-relevant abstractions (e.g., threads, TCP connections) that may not coincide with the needs of our accounting. Consequently we identify the following design principle that underlies our accounting solution: *our local monitoring must explicitly capture information pertaining to resource usage on behalf of CEs to allow accurate accounting by our collective inference.*
4 Resource Accounting Design and Implementation

Throughout this section, we first present the general ideas underlying our solution. We follow the description of each key idea with details of how we implement it within our prototype accounting system. Our prototype, called sAccount, comprises a cluster of up to 10 Dell poweredge sc1425 servers with 2GB RAM each and 1Gbps network. Each server runs a modified Xen 3.1.4 hypervisor within which we implement our local monitoring facilities. Additionally, a dedicated server receives monitored data from all others and runs our collective inference that yields the accounting information. Figure 3 presents the overall schematic for our sAccount prototype. All our source code as well as experimental data reported in this paper may be found at “http://csl.cse.psu.edu/saccount/”.

4.1 Local Monitoring

There are two key aspects to the local monitoring that we need to perform at each server: (i) recording information needed to identify the sets of used servers for each CE and the structure of resource accounting trees within each server, and (ii) identifying and recording information about resource principals and scheduling events of interest. In what follows, we discuss these two issues.

4.1.1 Identification of $S$ and $T$

General Design Considerations: We need to answer the following basic question: for a given pair of a CE $c$ and a server machine $m$, does $c$ make use of any resources on $m$ during the monitoring interval of interest? Recall from Section 2, that the real challenge in answering this question arises when $c$ uses resources on $m$ indirectly, i.e., when a shared service component $s$ running on $m$ consumes resources on behalf of $c$. How accurately the local monitoring on server $m$ can identify such indirect usage depends on its accuracy in recognizing the underlying causation (i.e., some activities of $c$ caused certain activities of $s$ which consumed some resources of $m$). When $c$ is only “one hop away from $s$, the presence of direct communication between $c$ and $s$ can yield this causation information. E.g., the front-end component of the shared database service in Figure 2 is one hop away from the CEs. However, identifying causation becomes trickier when the component $s$ is “more than one hop away” from $c$. An example of this is seen for the data store component in Figure 2 which is two hops away from the CEs. Solving this problem, in general, requires some form of statistical inference based on building a probabilistic model to capture this causation, and closely related examples can be seen in some recent work [3, 9].
Figure 4: Solution concept. Start and end of CPU accounting is determined by the arrival of messages and departure of response messages. As the thread $x$ of VM2 sends a message to the thread $A$ of VM1, the VM1 starts to account the CPU usage of thread $A$ to CE$_1$. This binding stops when thread $A$ sends reply back. CPU usage of thread $B$ is not charged to CE$_1$ inbetween. This requires us to be able to detect thread scheduling events.

**Realization in sAccount:** If a CE $c$ is only one hop away from a shared service component $s$, and is using its resources, the Xen hypervisor on the machine $m$ running $s$ simply recognize that $m \in S_c(t)$ if it observes an IP addresses belonging to $c$ on any of its incoming messages during the interval $[t, t + \Delta]$. To recognize a CE $c$ that is more than one hop away, we reply upon ideas from our prior work on vPath [27]. Very briefly, if one assumes that software components are constructed using a multi-threaded architecture where: (i) a given thread is only associated with acting on behalf of one CE at a given time, and (ii) all threads only employ synchronous communication, then the problem of identifying causation can be solved exactly (rather than only probabilistically as in the general case) [27]. A more general realization could employ statistical techniques mentioned above and is interesting future work.

**4.1.2 Identifying Resource Principals & Scheduling Events**

**General Design Considerations:** This aspect of local monitoring is concerned with collecting information about when a schedulable entity begins to use a resource on behalf of a certain CE and when it stops doing so. The local monitoring must record such information solely based on what the hypervisor can observe or discover about the resource principals on that server, and the events corresponding to their scheduling. For a server that is being directly used by a CE, this may be relatively straightforward. E.g., the hypervisor can simply use the per-VM scheduling information that it has access to. Additionally, one may consider collecting information to enable accounting of resources consumed by systems software (e.g., the hypervisor itself, privileged VMs that deal with significant portions of IO virtualization in many systems, etc.) on behalf of the CEs, similar to such accounting in resource containers [4].

For a server machine $m$ indirectly used by a CE $c$ (i.e., running a shared service component $s$ exercised by $c$), we need to identify CPU (de-)scheduling events within the software of $s$ that correspond to durations for which $s$ was using the CPU on behalf of $c$. Identification of any IO activities initiated during these same periods allows for accounting IO bandwidth usage by $s$ on behalf of $c$. Figure 4 gives illustrative examples of these ideas.

**Realization in sAccount:** How completely and accurately the ideal local monitoring described above can be realized depends crucially on certain aspects of the software architecture employed by the shared service component $s$ in question. In the sAccount prototype, we assume that shared service components employ the prevalent multi-threaded concurrency architecture. In this architecture, subsets of existing threads cater to each CE using $s$. Furthermore, each individual thread caters only to a single CE at any given time, although this mapping itself
can be dynamically changed by the application’s scheduling policy. With this architecture, it becomes possible to observe and record relevant scheduling events accurately from within the Xen hypervisor in the following manner: (i) context switching points within the VM hosting s correspond exactly to events when processing on behalf of a particular CE begins/end, (ii) context switches, despite being performed by the guest/VM kernel, trap to the hypervisor due to the paravirtualized nature of the Xen that we use, allowing its local monitoring facility to precisely record them, (iii) our causation establishment technique, described earlier, allows us to correctly keep track of dynamically evolving binding between threads comprising s and the CEs that they act on behalf of. Note that the reliance on paravirtualization for (ii) is not a significant shortcoming and can be overcome even in a system with a fully virtualized Xen hypervisor (e.g., written for Intel VT). An example technique for this is based on the following modification to such a hypervisor: the hypervisor uses the PRESENT bits in the PTE corresponding to the stack of the thread whose context switching it wishes to intercept.

Once CPU intervals used on behalf of a CE are identified, IO activities initiated during these are marked as corresponding to these CEs. Network-related monitoring is based on tracking the system calls that are related to the network activities. System calls such as READ/WRITE and RECV/SEND triggers network usage. Return bytes of these system calls are interpreted as the network bandwidth usage and accumulated to the corresponding CEs. Disk I/O-related monitoring follows similar principle as network accounting. The system calls to track are READ and WRITE. These two system calls are also used for the network reads and writes.

4.2 Collective Inference

Given the extensive information that our local monitoring gathers, collective inference for accounting CPU and network/disk IO bandwidth essentially boils down to simple aggregation of the resource usage information collected by various local monitoring units.

CPU Accounting: CPU accounting is done by measuring CPU cycle counts between start and end of the thread segments. Cycle counts are accumulated to corresponding CE’s CPU accounting variables. Thread segment that is not labeled is accounted as ‘unaccountable’ (See Figure 10(b) and Figure 16(a) for examples). The ‘unaccountable’ quantities tell us the possible range of errors in CPU accounting.

Network Accounting: Any network related activities between start and end of the thread segment is accounted to the current identified CE. We observe network-related system calls such as recv or send and add return byte sizes to determine how much of the network bandwidth has been consumed. Note that this quantity does not include any bandwidth consumption due to protocol-specific overheads such as retransmission try and various header/trailer portions added across the protocol stacks.

Disk I/O Accounting: Unfortunately, disk I/O accounting has limitations. Due to nondeterminism introduced by the page cache and block I/O handling mechanisms within the kernel, it is not possible to accurately identify the block traffics that are caused by each thread segments. From the application system calls, we can only know how many bytes are requested to be read or written. However, we cannot determine exactly which part of those translates to actual requests to the device. This forces us to resort to inference techniques from the information collected by sAccount techniques.

Disk READ traffic: When a thread issues disk READ I/O requests, there can be either page cache hit or miss. In case of hit, the system call latency is fast. In case of miss, the system call has to block until the data is fetched from the physical device. The latency of miss is significantly high. By measuring the latency of individual system calls, we are able to identify the disk READ I/O requests that triggered actual disk I/O traffic. We collect the number of reads (that missed the page cache) issued by each thread segment and use this ratio among CEs to divide the actual (read) block traffic observed at the storage device under the control of the hypervisor.
Disk WRITE traffic: Disk WRITE I/O requests do not exhibit latency difference between page cache hit and miss. All write I/Os hit page cache (unless page cache eviction is triggered) and destaged in bursts later. The sAccount is unable to precisely account the disk WRITE I/O requests for this delayed destaging and block I/O coalescing. Division by ratio as used in the READ case is not possible because the locality of each thread’s WRITEs may be different. In this case, we have no choice but to use inference techniques.

4.3 Implementation of sAccount

Our environment is based on the Xen [6] virtualization. We have employed para-virtualized xen on 30 dual-CPU blade servers divided over two racks. The distributed applications we have used in our evaluations are all hosted within individual virtual machines in these blade servers and they communicate over 1Gbps ethernet. In our environment we focus on performing resource accountings on three resource types (CPU, network bandwidth and disk bandwidth). However, our resource accounting is not limited to these resource. The basic principles of the technique, based on observing resource consumption at the thread granularity, can be extended to other type of resource that servers might use. For example, memory bandwidth, memory space and/or SAN networks/storage spaces can be subjected to sAccount frameworks.

Figure 3 depicts the overall architecture of sAccount and relevant components. At each physical hosts, we have modified Xen hypervisor to add functionalities for system call entry/exit interception, kernel thread switch interception and VM scheduling events. These information is delivered to Dom-0 and recorded through the modified version of xentrace. The output of xentrace from multiple hosts are transferred to a central location, labeled as ‘Accounting Node’ in the figure and this node runs the parsing and analysis algorithm to generate time series of resource usage per each physical hosts.

5 Evaluation

In this section we evaluate the efficacy of sAccount and compare it against a baseline called LR (described below). In the interest of space, we do not report results about the aspect of sAccount dealing with inferring and keeping track of dynamically changing sets of used servers for different CEs. Instead, we restrict our attention only to the most interesting aspects of the resource accounting tree within one particular server belonging to the shared service in question.

Our Baseline Accounting Technique (LR): Our baseline is based on a linear regression model relating the per-CE resource usage to the inbound network traffic volume from each CE. In order to account the CPU usage of a server to n different CEs, one can use the volume of inbound network traffic from each CE as X input and the CPU utilization of the server from TOP as the Y. Assuming the linearity between X and Y, we can solve AX=Y to find the coefficients. The coefficients can then be interpreted as the contribution of each client to the server’s CPU usage. For one measurement data point we can think of forming the following equation:

\[a_1 x_1^n + a_2 x_2^n + \ldots + a_n x_n^n = y_i\]  

where \(x_i^n\) represents the measurement of input traffic volume to the front-end of the shared service cluster induced from CE\(_n\) at time \(i\), and \(y_i\) the aggregate resource usage measured by system utility functions of target resource (e.g. CPU utilization by \(\text{top}\)). At time \(i\), the coefficients \(a_1, a_2, \ldots, a_n\) is interpreted as how much each CE has contributed to the resource usage. Therefore, negative coefficient values are undefined.

5.1 Accounting Accuracy for a Synthetic Shared Service

Experimental Setup: Fig. 5 shows the design and configuration of a synthetic shared service we employ. We use a two-tiered design for the shared service with the front-end acting as a caching tier. Cache misses in the front-
tier result in work generated at the back-end. Multiple clients send requests to the front-end during long-lasting sessions and correspond to our synthetic CEs. We define operation types offered by the server whose resource consumption we extensively measure offline to construct the “ground truth” about their resource needs.

**Experiment Design and Key Findings:**

**Bursty vs. Non-Bursty Workload:** Figures 6(a)-(b) compare the effect of burstiness/variance in workload on the accuracy of CPU accounting. For different values of the average request rate imposed on the shared service by a group of three chargeable entities $CE_1, CE_2, CE_3$ (which create different CPU utilization levels at the server), we pick a “non-bursty” scenario where the requests are uniformly spaced in time, and a “bursty” scenario where the request inter-arrival times follow lognormal(0,1.0) distribution. We find that the efficacy of LR varies depending upon the extent of variation within the imposed workload. This is in line with results known in existing work that find non-stationarity in workloads useful for certain kinds of prediction and modeling [26]. Intuitively, better accuracy is achieved with bursty workloads because the higher variety/dynamism in the input data supplies more information to LR; we expect this basic insight to apply to any statistical inference technique for accounting. For a less bursty workload, a large part of the input data may be redundant and not offer new information to an inference technique. On the other hand, by virtue of its direct measurement of relevant phenomena, sAccount is able to achieve accurate accounting that is robust to changes in such workload conditions. We also find the accuracy of LR to be sensitive to the overall CPU utilization level at the shared server, although we do not see a clear pattern. sAccount performs well in all utilization regions offering less than 1% error, which may be particularly desirable in the high utilization regions.

**Caching/Buffering:** We find that caching and buffering - both valuable and prevalent performance enhancement techniques - affect the accuracy of accounting of a technique like LR. As is well-known in general, a cache within or in front of a service can destroy/distort correlations between its incoming request/traffic events and the workload imposed on its underlying server in complex ways. Buffering of requests/traffic can also have a similar effect by modifying the time lag between an event (e.g., the issuance of a request) and its cause (e.g., the actual servicing) in complicated ways. We carry out experiments where we vary several factors affecting the degree and nature of caching within the front-tier of our shared service (see Figure 5): request size (fixed or varying), read/write ratio (from 10:1 to 1:1), temporal locality (non-existent to very high), and the extent of common/overlapping content requested by the chargeable entities. Of the large parameter space, we present results for the following four kinds of workloads imposed on the shared service: (i) $w_1$, with a choice of factors that we expect to offer minimal caching gains, (ii) $w_2$, with a choice of factors that we expect to offer significant caching, (iii) $w_3$, moderate gains between the last two, and (iv) $w_4$, a workload that uses these three in succession.

![Figure 5: Design and configuration of our synthetic shared service and the CEs exercising it.](image-url)
Figure 6: Impact of burstiness and shared service resource utilization on the accuracy offered by sAccount versus LR. We use our synthetic shared service along with three chargeable entities. We compare the percentage error in CPU accounting for sAccount and LR. We label the errors for our three chargeable entities with LR as $CE_1$, $CE_2$, and $CE_3$, respectively, and label their average as “LR Average.” In all cases, the accounting information offered by sAccount shows less than 1% error (we plot the average of error for the three CEs).

Figure 7(a) shows results for the accuracy of accounting the network bandwidth used for communication between the front-tier (cache) and back-tier of our shared service. The chargeable entities impose a bursty workload (as described earlier) that imposes an average CPU load of 50% on the server hosting the front-tier. The x-axis of the Figure 7(a) indicates the length of data points fed into the LR. For example, ‘1 min’ means that we have used 60 data since we collected measurements every one second. If the total run is 20 minutes, then we perform 20 LR computations for ‘1 min’ case and 4 LR computations for ‘5 min’ case. The longer data we use for LR, the better the accuracies become. The graph shows that caching also has large impact to the accuracy. Accuracy gains from the burstiness of the workloads can be easily offset if application happens to employ some form of caching structure internally.

Varying Number of Chargeable Entities: Figure 7(b) shows the effect of the number of chargeable entities on the accuracy of LR and sAccount. We observe that the accuracy of LR (both average and variance) deteriorates as the number of chargeable entities grows whereas the accuracy of sAccount is unaffected.

Summary of Key Findings: To summarize, we find that the efficacy of LR relies upon both the quality of data it gathers as well as the presence/extent of correlation between its inputs and outputs. Even when accurate data can
be obtained (as with our implementation of LR), several factors including (i) inherent workload properties (e.g., variance, temporal locality, intensity), (ii) system mechanisms and algorithms (e.g., caching or buffering), and (iii) environmental conditions (e.g., degree of resource interference from other software) might affect such correlation and affect the accuracy of the accounting technique. We find empirical evidence that, owing to its ability to directly measure relevant phenomena accurately, sAccount is robust to such effects, and offers high-accuracy accounting information across a wide range of operating conditions.

5.2 Accounting for Real-world Services

In general, we cannot expect to determine a real-world application’s actual resource usage on behalf of different chargeable entities without resorting to extensive application and OS instrumentation. Consequently, unlike for our synthetic shared service, we cannot obtain/present a direct comparison of the efficacy of our techniques, i.e., distance of the accounting information offered by sAccount versus that offered by LR from the “ground truth.” We present results for the accounting of the most bottlenecked resource for the shared service, which we find to be CPU cycles for MySQL and network bandwidth for HBase.

5.2.1 Clustered MySQL as the Shared Service

Experimental Setup: Figure 8 shows the set-up of our MySQL cluster that is used as a shared service by three CEs. Two of these CEs use the TPC-W benchmark [25] to generate workload for the database, while the third CE uses RUBiS [24]. The cluster consists of a front-end SQL node that interacts with the chargeable entities, three data nodes, and a management node; each node is hosted within its dedicated server. One interesting aspect of the cluster’s operation is that even in the absence of any workload imposed by the CEs, a large number of small messages are exchanged between all pairs of nodes within the cluster. These messages are for liveness check. The CEs house separate/non-overlapping data within the database which is spread across the three data nodes, and the cluster a replication degree of 1.

Experiment Design and Key Findings: Given exact accuracy numbers are elusive, we compare the efficacy of sAccount and LR in the following online resource control situation: we wish to ensure that when the aggregate workload imposed upon the MySQL cluster causes its server CPUs to saturate, we identify the contribution of various CEs to this “overload,” and then enforce targeted CPU throttling only to the CE causing the overload.
Table 1: Description of how the workload imposed by the three CEs is varied over the course of our experiment with the MySQL cluster as our shared service.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time Window</th>
<th>Workload</th>
<th>Top User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>0-400s</td>
<td>All 3 CE generate light load</td>
<td>CE_2</td>
</tr>
<tr>
<td>Phase 2</td>
<td>400-600s</td>
<td>CE_2 starts to issue CPU-heavy requests</td>
<td>CE_2</td>
</tr>
<tr>
<td>Phase 3</td>
<td>600-1200s</td>
<td>CE_2’s workload overwhelms CPU, load increases every 100s</td>
<td>CE_2</td>
</tr>
</tbody>
</table>

We implement a CPU policing mechanism within the Xen hypervisors of the MySQL cluster servers which manipulates the rate at which timer interrupts are delivered [14] to the guest VM only when the thread serving the CE causing the overload is to be scheduled. Our choice of this policing mechanism is only for demonstration purposes, and in practice a more sound technique would be desirable.

We configure our CEs to impose a dynamically changing workload (consisting of three phases) on MySQL as described at a high-level in Table 1. In phase 1, all CEs generate a low-intensity workload, whose aggregate does not saturate the MySQL servers. During phase 2, starting at t=400s, CE_2 starts issuing more CPU-intensive
requests. We are interested in observing how LR and sAccount handle this sudden change of behavior. Finally, in phase 3, starting at t=600s, CE₂ issues continually increasing workload that causes the CPUs to saturate. Here are we are interested in observing how our simple resource control performs based on the accounting information offered by LR and sAccount.

Since we do not have precise knowledge about true resource consumption, we engineer the workloads so that the CPU consumption imposed by the CEs are significantly different from each other, allowing us to rank their contributions without ambiguity. For example, we make the CPU consumption of CE₂ much larger than other starting at t=400s so that other CEs cannot be mistaken as heavy CPU consumers. We begin by taking an in-depth look at the CPU accounting information offered by LR and sAccount at one of the MySQL servers (SN) and how it evolves during phases 1-3 (results for CPU accounting of other MySQL nodes are qualitatively similar and we do not present them in the interest of space).

In Figures 9(a),(b), we depict the inputs for LR (per-CE network traffic and aggregate CPU usage at SN’s server) and its output (accounting information for each CE), respectively. These figures are helpful to consider in combination with the following discussion of LR’s accounting and its comparison with sAccount.

Figures 10(a),(b) show CPU accounting for SN’s server as carried out by LR and sAccount, respectively. We use a ”stacked” representation, where the area under the curve corresponding to a CE represents the CPU usage charged to it. During phase 1, both LR and sAccount produce correct rank orders of CEs, although LR slightly overestimates the CPU consumption for CE₂. However, during phase 2, LR starts to report incorrect rank order: it determines CE₃ to be the cause of the increased CPU usage. Upon investigating the reason for this
mistake by LR, we find the following. While CE\textsubscript{2} issues CPU-heavy requests and waits for MySQL’s response, CE\textsubscript{3} continues to issue requests at a relatively high rate that are not CPU-heavy. However, the higher rate of requests coming from CE\textsubscript{3} causes LR to infer spurious positive correlation between CE\textsubscript{3}’s requests and SN’s CPU usage. In fact, LR is unable to correct this throughout phase 2.

As we show in Figure 10(b), besides correctly identifying the correct rank order in its accounting, sAccount also reports what portion of the CPU usage of SN’s server it finds unaccountable. This amount indicates that sAccount’s algorithm was unable to charge the given thread’s resource usage to any of the chargeable entities because no direct association was found. This can happen when some thread is spawned independently of input requests from the chargeable entities and performs maintenance jobs. Or, it could be due to the nature of the thread that is created to service other running threads. In any case, sAccount provides this resource usage to the user and it up to the user to divide up among chargeable entities. The most reasonable division would be to divide the ‘unaccountable’ portion according to the proportion of resource usage by each chargeable entity within that time window.

During phase 3, starting at t=600s, CE\textsubscript{2} starts to saturate the CPU by drastically increasing the workload it imposes as described in Table 1. As portions of Figures 10(a) and (b) for this phase show, LR continues to perform incorrect accounting. This has a detrimental effect on our CPU policing based resource control. Figure 11 shows the change of response time for CE\textsubscript{3} whose RUBiS application is accessing the shared MySQL Cluster service. Starting from time 600, the response time increase. We have set the response time of 300 ms as the initial warning level and 600 ms as the SLA violation level. The CPU saturation caused by CE\textsubscript{2} continues to degrade the response time of RUBiS and eventually it violates the SLA. Since LR determines that CE\textsubscript{3}, not CE\textsubscript{2}, is the source of overload (see Figure 10 (a)), CE\textsubscript{2} is not marked for any counter actions. However, sAccount is able to identify true cause of the overload and, starting at time 730, it initiates the CPU throttling for CE\textsubscript{2}. Figure 11 indicates that the moving average of response time under the control of sAccount is able to contain the response time below the SLA limit. This demonstrates one promising capability of sAccount (which is the thread-level monitoring technique) in critical resource managements of such shared resources.

5.2.2 HBase as the Shared Service

Experimental Setup: Our second real-world shared service is HBase, a key-value storage system offering an open-source implementation of Google’s Bigtable [8], that has significantly different resource usage characteristics from a database such as MySQL. An HBase cluster consists of “region servers” and the HBase “master”
Figure 12: Our setup for using HBase as a shared service. Our CEs are based on client programs that use the YCSB workload generator.

that manages these region servers. HBase stores its data in a Hadoop cluster, and this is a great example of a shared service (HBase) relying upon another shared service (Hadoop) to cater to the needs of CEs using it. The region servers act as in-memory caches for the contents of the data nodes. A region server employs an HDFS client [15] to communicate with the Hadoop cluster that stores data persistently. HBase employs Zookeeper [16] for coordinating distributed operations and locating region servers. We configure our HBase with a single region server. HBase operation involves significant data transfer from the data nodes to the region server, whereas the CPU load imposed by most requests is small as most requests are for simple data retrieval or inversions. Since the network bandwidth available to the region server becomes the bottleneck resource well before the CPU, we highlight accounting results for network bandwidth more. Our HBase caters to requests from two CEs derived from the YCSB workload generator [12], that we refer to as CE_A and CE_B.

Experiment Design and Key Findings: We run an experiment lasting 500 seconds, during which the loads offered by CE_A and CE_B are varied as follows: (i) during t=0 to t=100s, both CE_A and CE_B generates identical workloads which contains 5% update requests, (ii) at t=100s, CE_B changes to a read-intensive mode with good temporal locality, which incurs high hits in the region server causing its CPU usage to increase proportionally with the network traffic sent to CE_B, (iii) at t=200s, CE_A starts to issue CPU-intensive insert-type requests that cause the CPU usage at the region server to increase. Figure 13 shows the network traffic size inbound to the region server from the two CEs under the described workload scenario.

Figure 14 shows the profiling result of individual runs for both CE_A and CE_B chargeable entities. Unlike Mysql Cluster where there can be several different request types each having varying resource demands in terms of CPU and Network, HBase seems to behave in a much simpler way. HBase’s request types in terms of resource consumption are not diverse, because HBase uses simple key-value storage that usually does not carry out complex logic functions such as join and sub-query. As a result, for both CPU and network, they show very similar trends with client input network.

Summary of Results: Figure 15 (a) and (b) shows the result of accounting the network bandwidth by LR and sAccount at the region server of HBase. The input to the LR are two time series of inbound network traffic from two chargeable entities as shown in Figure 13. We have configured LR to use 100 second-long data as an input length in this HBase’s resource accounting, producing no accounting results for the first 100 seconds of the run. The accounting results from sAccount is presented in Figure 15 (b). In both of the stacked graphs, the red area (the
Network Traffic (KB)

CE_A goes read-intensive

CE_B goes update-intensive

Time (Seconds)

Figure 13: Evolution of network traffic that are incoming to the region server from two CEs during the run. Both CE_A and CE_B start out by sending similar requests to HBase during t=0 to t=100s, implying the network bandwidth and CPU usage of the region server should be accounted equally to them for this period. CE_A changes its behavior at t=100s, whereas CE_B changes its behavior at t=200s.

CPU Utilization (%)

CE_A
CE_B

(a) CPU usage at the region server for two CEs.

Network Throughput (KB)

CE_A
CE_B

(b) Network input traffic from two data nodes to the region server.

Figure 14: Profiling measurement of CPU and Network resources at the region server for CE_A and CE_B. These are obtained from individual (not combined) runs of each chargeable entities. They are intended to serve as an estimate of true resource usage quantity when analyzing the performance of resource accounting results.

top most area) corresponds to the portion of network bandwidth used by CE_B and the blue area (the second area from the top), the portion used by CE_A. From the Figure 14, we know that the network bandwidth usage of CE_A should significantly drop from time 200, and thus making CE_B the heavier consumer of the network bandwidth. We observe that the result of LR unfortunately does not reflect this true values. LR reports that CE_A remains to be the dominant consumer of the network bandwidth throughout the run. In contrast, sAccount correctly reflects this resource usage by CEs. (See the size of red area in Figure 15 (b) after time 200). The misjudgment by LR can be explained in terms of caching effects. Since time 200, the input network traffic from CE_B to the region server doubles until the end of run (See Figure 13) whereas the incoming traffic from the data nodes increases by only 20% (See Figure 14(b)). We believe this is due to the internal caching at the region server and HBase documentation supports this conjecture. Earlier in Figure 7, we have seen that caching can impact the performance of LR.

In addition to the network accounting results, we also present some of the selected accounting results using sAccount. Figure 16 (a) shows the CPU accounting at the region server for CE_A and CE_B. According to our
preplanned workload scenario, CE$_B$ should consume more CPU than CE$_A$ after time 200 sec. The accounting result indicates this behavior. Notice the significant portion of unaccountable CPU usage at the bottom region of the stacked graph (orange color). It includes CPU cycles consumed for communicating with the Zookeeper, HBase master and Hadoop Namenodes. Especially, we have noticed two high peaks at around 80 sec and 470 sec. By studying the HBase documentations, we have concluded that these would most likely be due to the I/O compaction at the region server. These nondeterminism disturbs the correct operation of LR whereas our sAccount correctly identifies them as not being correlated to chargeable entities. Figure 16 (b) presents the accounting result of network usage in one of the datanodes. We present this result here in order to provide evidence that our sAccount can perform resource accounting at the node that lies “more-than-one-hop” away from the chargeable entities. This shows that our implementation can successfully deliver the causality of messages to across nodes and make use of them into resource accounting.

6 Related Work

Earlier works by Banga et. al. [4] have addressed the issue of correct resource accounting within a single host. They introduced new abstraction, called resource containers, to be used as a resource principal within the kernel. Distributed resource container [29] is an extension of the resource container to the distributed environment in which local resource containers are bound together by exchanging identifiers in order to coordinate the resource
consumption across hosts. In their work, the goal was to throttle the energy consumption per applications. Our work advances this thread of research by enabling resource accounting at various locations within the distributed application hierarchy by exploiting the message causality tracking technique and thread-level monitoring capability.

Recall that there are two aspects to resource accounting solution: *local monitoring* and *collective inference*. Some monitoring techniques do not require any modification to application software, and we label all of them as non-intrusive. These techniques span a spectrum of the effort involved in modifying the underlying systems software. Among the least intrusive techniques are popular user-level monitoring tools such as `top`, `iostat`, `vmstat`, `sar`, `netstat`, etc. These tools either rely upon OS system calls or read certain OS-provided information (such as `/proc/stat`) to find resource usage. Some techniques insert hooks within the systems/runtime software for data collection. While still non-intrusive according to our classification, they entail different amounts of additional effort in their design and use. For example, the tool OProfile [17] requires insertion of a kernel module, or kernel recompilation with reconfiguration. Chopstix [7] adds a data structure, `sketches`, to the kernel in order to monitor low-level OS operations such as page allocation, mutex/semaphore locking and CPU utilization. Kprobes [1] allows you to insert probes to the kernel functions or addresses. Since it induces breakpoint exceptions, it is known that performance degradation is high. Using Kprobes may require turning `CONFIG_KPROBES` and other configurations on and rebuild, depending on the Linux distributions. It is intended to be used for kernel debugging. Xenprobes [20] can be used to inject breakpoints into entry and exit point of

---

**(a) CPU Accounting by sAccount at the region server. This is a stacked graph.**

**(b) Network accounting by sAccount at data node. This is a stacked graph.**

Figure 16: Resource accounting by sAccount at various nodes of HBase. The result (b) provides the proof of sAccount’s capability to perform resource accounting at multiple hops away from the front-end of the shared service. Note that data node of HBase does not have a direct communication with any of the chargeable entities.
any kernel function within the guest domain, similar to Kprobes. Although Xenprobes is designed with test and debugging in mind, sAccount can certainly make use of Xenprobes to collect richer monitoring information. Unfortunately, the code is not available to public as of now, preventing us from neither adopting it nor investigating the feasibility.

An intrusive technique, on the other hand, requires modifying the application itself. The additional information resulting from these modifications often allows more accurate/detailed monitoring. However, this comes at the cost of added programming effort of modifying the application (which may be difficult or even impossible in many scenarios) and a possible run-time slowdown. For example, IBM ARM [18] requires recompilation after instrumenting the application with certain calls for monitoring the travel path of user requests through servers.

Monitoring itself is seldom the final goal and there are usually domain-specific higher-level goals (e.g., while the goal of inference in this work is resource accounting, frequently occurring goals in existing work are capacity planning [30, 31], debugging [2, 22, 23, 10], performance management [13, 18, 21]). Once monitoring data is collected, the next step is to apply suitable inference techniques that process this data to derive information needed for achieving such goals. The body of work on inference techniques is, of course, vast (see [31, 30, 26, 11] for some surveys) and spans the entire spectrum of statistical sophistication. For example, non-intrusive tools mentioned earlier that needs to interpret application-specific logs (e.g., access.log for the popular apache web server) can be considered as a simple inference technique. On the other hand, application of TAN model to the classification problem of SLO violations [11] falls into the group of sophisticated inference techniques.

7 Conclusion

We studied the problem of resource accounting within an IT platform that offers shared services to its users. We designed and implemented a prototype of our accounting solution called sAccount on a cluster of servers running the Xen hypervisor. Using a mix of synthetic and real-world shared services, we demonstrated the efficacy of sAccount in carrying out accurate accounting of CPU and IO bandwidth resources. Furthermore, we found that sAccount was able to consistently provide high-accuracy accounting under a wide variety of conditions under which a baseline representing the state-of-the-art performed poorly.

References


