SPIN: Service Performance Isolation Infrastructure in Multi-tenancy Environment

Xin Hui Li, Tian Cheng Liu, Ying Li, and Ying Chen

IBM China Research Lab, Software Park
Beijing 100193, China
{lixinhui,liutc,lying,yingchen}@cn.ibm.com

Abstract. The flourish of SaaS brings about a pressing requirement for Multi-tenancy to avoid dedicated installation for each tenant and benefit from reduced service delivery costs. Multi-tenancy's intention is to satisfy requests from different tenants concurrently by a single service instance over shared hosting resources. However, extensive resource sharing easily causes inter-tenant performance interference. Therefore, Performance isolation is crucial for Multi-tenancy environment to prevent the potentially bad behaviors of one tenant from adversely affecting the performance of others in an unpredictable manner and prevent the unbalanced situation where some tenants achieve very high performance at the cost of others. Current technologies fail to achieve the goals of both performance isolation and resource share. This paper proposes a Service Performance Isolation Infrastructure (SPIN) which allows extensive resource sharing on hosting systems. Once some aggressive tenants interfere with others' performance, SPIN gives anomaly report, identifies the aggressive tenants, and enables a self-adaptive moderation to remove their negative impacts on others. We have implemented SPIN prototype and demonstrate its isolation efficiency on the Trade6 benchmark which is revised to support Multi-tenancy. SPIN fits industry practice for a performance overhead less than 5%.

Keywords: Multi-tenancy, performance monitoring, resource accounting and management, byte code instrumentation.

1 Introduction

Software-as-a-service (SaaS) [1] permits customers to consume software applications in a hosting mode as an emerging software delivery model with the capabilities of lowering total cost of ownership, fast enablement, and seamless scale-up per business needed, especially by Small and Medium Businesses (SMB). SaaS is typically associated with "multi-tenant architecture", which is a prerequisite for a SaaS application [2, 3]. Traditionally, there would be only one instance of an application running on a server, and this instance would only serve one customer, organization, or company (tenant). In the SaaS world, giving each tenant a dedicated server is a huge waste of resources and service providers want to put as many tenants on the same server as possible. Multi-tenancy aims to enable a service environment that user requests from different tenants are served concurrently by the least amount of hosted service

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 649–663, 2008.
© Springer-Verlag Berlin Heidelberg 2008

instances running on the shared hardware and software infrastructure. It requires deployment of a much smaller infrastructure, in contrast to having a dedicated installation for each tenant, which bring in a number of benefits including improved profit margin for service provider through reduced delivery costs and decreased service subscription costs for clients.

Multi-tenancy has two different maturity patterns: the first pattern supports each tenant with its dedicated service instance over a shared hardware, Operating System (OS) or a middleware server in a hosting environment whereas the second pattern can support all tenants by a single service instance over shared hosting resources. The second pattern is more consistent with the intention of Multi-tenancy. In the environment of pattern two, the tenant would naturally desire to access and use the service as if there were dedicated ones. However, extensive resource sharing easily causes inter-tenant interference on performance. To evolve from pattern one to pattern two and achieve more efficient Multi-tenancy hosting, performance isolation is crucial to prevent the potentially bad behaviors of one tenant from adversely affecting the performance of others in an unpredictable manner and prevent the unbalanced situation where some tenants achieve very high performance at the cost of others.

At present, virtualization technology is used to enable the isolation needed by Multi-tenancy and isolation management [4, 5, 6]. Some directly depend on Virtual Machines to create service hosting environments that provide logical boundaries between tenants. Although these works can help to adapt current software and hardware for Multi-tenancy with the least cost, virtualization technology belongs to the pattern one and is not able to cater for the demands of Multi-tenancy. They restrict resource sharing between different tenants and cause additional management cost.

There are improvements on the original adoption of virtual machine to reduce the amount of exclusive and dedicated computing resource. SWSoft's Virtuozzo product [7] is an example of that technology. This architecture allows service partitions to be created and configured differently from one another. Although it is tremendously valuable to SaaS hosters for optimizing their machine allocation, especially for these ISVs having the same system requirements, virtual service partition in general is not an effective method for Multi-tenancy, since it needs a different service instance for each partition and does not support multi-tenants to share all resources of hosting platform.

Normally, hosting systems have sufficient resources to meet basic requirements of tenants, but they can not provide enough resources to meet everything that every tenant might want with variations on the workload from some aggressive tenants. The performance experienced by the workload from a given tenant suffers from such resource unavailability. We call this situation as Instable state where the hosting systems have acted their processing capability to the full extent, even exhaustingly to crash, but can not satisfy requests of every tenant. With this in mind, we advocate a service performance isolation infrastructure (SPIN) which achieves efficient isolation and extensive resource sharing simultaneously. SPIN makes anomaly detection for instable state, identifies aggressive tenants, and enables the multi-tenancy system to self-adaptively remove negative impacts of the aggressive tenants on others. SPIN has been implemented as a Plug-in, independent of server and service implementations. The practice of SPIN in the revised Trade6 [8] demonstrates its accuracy of anomaly report, effectiveness of performance isolation and little perturbation to the running

service (less than 5% performance overhead). The implementation and experiment of SPIN is made based on Web service, a specific kind of popular service.

The rest of this paper proceeds as follows. Section 2 studies the requirements on performance isolation infrastructure and design of SPIN. Section 3 describes our prototype implementation. Section 4 evaluates the effect of SPIN with business service benchmark. Finally, Sections 5 and 6 discuss related work and conclusions.

2 Service Performance Isolation Infrastructure (SPIN)

In this section, we first outline the requirements of performance isolation in the complex multi-tenant environments. We then examine how to meet these requirements in our design.

Isolation. The infrastructure should prevent aggressive tenants from interfering with others. Request from different tenant will trigger different execution in the backend modules because the instances of these modules are shared among all the tenants sharing the same suite of resources. For this propose, it is necessary to account resource usage for different tenants during their access the shared modules to understand the performance factors of hosting platform. It is not permitted for some aggressive tenants to encroach resources and cause others' performance decreasing.

Efficiency. The infrastructure should maximize the overall utilization of resources on hosting platform, which is the intention to induce Multi-tenancy. This might be achieved by placing a loose upper bound on resource usage of different tenants. Pre-reservation of resources [4, 9, 10] can not satisfy this point. This goal is motivated by the fact that systems are likely to have sufficient resources to meet basic requirements of tenants, but they probably do not have sufficient resources to meet everything that tenants might want.

Self-adaptability. The infrastructure should not require user intervention or manual tuning. This goal is motivated by the large number of tenants which a hoster may serves as well as the wide range of services and system configurations likely to be involved.

To meet the above requirements, three principal functionalities, performance anomaly detection, system monitoring and adaptation decision, are provided in SPIN:

The ***anomaly detection functionality*** of SPIN is responsible for signaling the occurrence of instable status in the execution environment. The anomaly detection facility identifies and analyzes any significant variations happening on the performance metrics, especially the variations those might potentially affect the system's behavior in the immediate future. In this way, the infrastructure can preemptively adapt the system to prevent predicted performance problems from actually occurring.

Although some work [11,12,13,14] has been made on the performance of service system in the passed years, there are primarily two points preventing these technologies applied in our anomaly detection. One point is their dependency on threshold. It is difficult to give threshold values in a deterministic and automatic way since the presence of Multi-tenancy brings out the complexity and randomness into present service system. The other point is that they pay no attention on the prediction of

instability. In practice, it is more important to predict and avoid problems on resource usage in the near future than to remove the negative impact already caused.

The ***monitoring functionality*** is responsible for collecting runtime data from the service components and their execution environment. On one hand, the data are used for detecting performance anomalies in service hosting platform. On the other hand, resource consumptions of each tenant must be accounted to identify which tenant is aggressive and whether the system serves that tenant with effective resource usage.

Under the complex Multi-tenancy environment, existing resource accounting technologies are impracticable. They [15,16,17,18] uses processes or threads as the accounting unit, while the real-life service applications run with multiple threads' concurrent execution. An individual thread may traverse various modules of services in the system. Further more, in the widely used thread pool one thread at one time serves one service and it will serve another service soon. And one instance of service serves several tenants and its execution is unaware to underlying thread or process.

The ***adaptation decision functionality*** of the infrastructure applies optimal solutions to those detected or predicted performance problems. If the state is instable, we identify the aggressive tenants from others based on their abnormal resources usage. Optimal moderation policy will be adopted automatically to isolate the negative effect and prevent the interference with other tenants.

SPIN starts System monitor and anomaly detection from the beginning. Anomaly detection is made on the data got by system monitor and reports anomaly state of the single service instance near before instability's happening. Simultaneously, the resource consumptions on the hosting system are also monitored and accounted on each tenant's behalf during their service usage. Once anomaly report is given, the resource consumption trend of different tenants is analyzed to identify which tenant's behavior causes the instability. Then, adaptive decision function is activated and moderation policy is executed automatically on the identified tenant to remove its negative effect on the performance of other tenants. These main functionalities are discussed in details over the following sections.

2.1 Anomaly Detection Model

For SPIN, we propose a new model to achieve sensitive anomaly detection. Wallace has proven in his paper [4] that the relationship between the mean arrival rate of service requests and the mean service rate is proven tightly correlated with the stability of system. If the difference between the arrival rate and service rate is positive, the system is stable and the request arrivals do not beyond the processing capability of the system; otherwise, the system is instable. In this model, the value of difference is used to execute the anomaly detection instead of the dependency on any threshold. Besides Queueing Theory, the model adopts a combination of Discrete Wavelet Transform (DWT) multi-resolution analysis [19] and Autoregressive (AR) model [20] to make a prediction on the difference value. It is named with WAQ accordingly. In this way, instability of service system in the short term is predictable, which provides convenience for system moderation and is favorable to health maintenance. Followings are the detailed calculation process.

For model WAQ, the following data items are got by monitoring as the inputs:

$$(c_i, t_i) \qquad i = 0, 1, 2, ..., k \tag{a}$$

Where c_i represents the number of requests that enter the service system at sampling time t_i ,

$$(b_i, t_i) \qquad i = 0, 1, 2, ..., k \tag{b}$$

Where b_i represents the number of requests that have been processed at sampling time t_i ,

From the above data items (a) and (b), we can derive the pure increasing rate by Equation (1):

$$x_i = \frac{c_i - b_i}{t_i - t_{i-1}}, i = 1, 2, 3, ..., k \tag{1}$$

Let series *{X_i} i= 1,2,..,k* be the input of prediction model. In the prediction model, the original discrete series of pure increasing rate is firstly decomposed into approximate series and several detail series. The result of single branch reconstruction of each decomposed series is more unitary than the original series in frequency, and it can be easy to predict by autoregressive method. At last, the prediction value of increasing rate can be obtained by synthesis of each reconstructed series' prediction result [20]. The prediction process works as follows:

Firstly, $a_{j,m}$ and $d_{j,m}$ $(1 \le m \le k)$ are got respectively by Mallat Algorithm [21] as the approximate series and detail series at resolution level j. In this way, the original series $\{X_i\}$ $i = 1,2,..,k$ is transformed into a set of stationary series and AR model is a powerful tool for prediction of such series [22]. The predicting expressions can be expressed as Equation (2) and (3):

$$a_{j,k+1} = \sum_{i=1}^{p} \phi_i \, a_{j,k-i+1} \qquad j = 1, 2, ... \tag{2}$$

$$d_{j,k+1} = \sum_{i=1}^{p} \phi_i \, d_{j,k-i+1} \qquad j = 1, 2, ... \tag{3}$$

Where $a_{j,k+1}$ and $d_{j,k+1}$ are the prediction values of approximate series $a_{j,m}$ and detail series $d_{j,m}$ at resolution level j respectively. ϕ_i is the corresponding coefficients of AR(p). Then, $\tilde{a}_{j,k+1}$ and $\tilde{d}_{j,k+1} (i = 1, 2, .., j)$ are reconstructed respectively from $a_{j,k+1}$ and $d_{j,k+1}$ by Mallat Algorithm.

As Equation (4) presents, $\tilde{x}_{k+1}$ is derived as the prediction value of time t_{k+1} from the original increasing rate series $\{X_i\}$.

$$\tilde{x}_{k+1} = \tilde{d}_{1,k+1} + \tilde{d}_{2,k+1} + ... + \tilde{d}_{j,k+1} + \tilde{a}_{j,k+1} \tag{4}$$

If $\tilde{x}_{k+1}>0$, the system will be instable at the time $t_{k+1}$and the anomaly report is sent out.

2.2 System Monitoring

System Monitoring function of SPIN gets inputs to feed Anomaly Detection Model and accounts resource usage on behalf of each tenant. When Anomaly Detection Model triggers an anomaly report, aggressive tenants will be identified according to the characteristics of their resource consumption.

Model inputs are got by accounting how many requests come to the server and are processed every sampling interval. Following sections introduce the design of SPIN on resource accounting for each tenant.

2.2.1 Resource Consumption Accounting on Behalf of Tenants

In SPIN, we adopt a new mechanism to monitor resource usage within the service execution and allow the proper assignment of resource usage to tenants. The design of monitoring mechanism is guided by two key constraints. The first is that the monitor function must keep active during the normal service execution to provide the ability of resource consumption tracking and therefore have minimal discernible impact to the service's runtime performance. As for the resources we focus, CPU and memory are our primary focus. The second constraint originates from the fact that hoster platform is dynamically deployed with object code service applications. This means that there is no opportunity to use the source code to implement the function of monitoring. Meanwhile, we do not make any modification on service container regarding the applicability of SPIN in practice.

The whole monitor is event based. Two kinds of events are triggered and listened: one is triggered by entry or exit of a service boundary, the other event is produced when some resource is consumed, like the allocation of memory.

For every boundary change event, present executing service is put into a stack, named accounting stack, for resource accounting. The accounting stack is designed to easily find present running tenant on which resource is accounted. Otherwise we have to get the whole stack of thread and walk it down to find the present executing service and tenant. Massive performance overhead will be induced into service running to frequently get stack data and walk stack.

For every resource consumption event caught, the top unit is gotten from the accounting stack as the owner of the resource consumption. In other words, the accounting stack is active at the time of resource being consumed and used as the context within which to determine accountability. Section 3 discusses the specifics of the implementation of accounting stack and accounting process.

2.2.2 Aggressive Tenants Identification

After the anomaly report, it is needed to identify aggressive tenants who are apt to consuming resources faster than others. They consume resource with a trend of growth, even snatch resource from other tenants and cause the degradation of others. The identification is made offline for performance consideration.

For each tenant, all the accounting events are recorded. Based on these events, the time series of each tenant's resource consumption is provided to users. Because the

absolute volume of resource consumption of different tenants can fall in different scale ranges, the percentage of each tenant's consumption in all is calculated and used to execute clearly comparison among all tenants.

We rank tenants according to the ratio of resource volumes between two consecutive accounting events (For convenience of description, the events are presented by t_i and t_{i-1}, i>=1, and the volumes of the two events are presented by Vt_i and Vt_{i-1}.), find the tenants with maximum ranks and report the tenant as aggressive ones. Because resource usage can be different from one request to another, the ratio value of one tenant may fluctuate up and down. We uses a decay factor f, where $0 < f < 1$ to adjust for the jitter. We considers only those tenants whose volumes satisfy $Vt_i > (1-f)*Vt_{i-1}$ on consecutive accounting events as potential candidates. The decay factor keeps the ratio value of tenants that shrink a little in this accounting time, but which may ultimately be growing. We find that the decay factor is increasingly important as the size of the resource accounts decreases. Choosing the decay factor balances between too much information and not enough.

To rank tenants, we firstly calculate the growth factor (G) of each accounting event as $Gt_i = Pt_i*(R-1)$, where P is the number of accounting event (t_i) that Vt_i has been potentially growing and R is the ratio of Vt_i at this event and Vt_{i-1} at the previous event such that R>1, since R>1, G>0. Each tenant's rank Rt_i is calculated by accumulating the growth factors G over several accounting events such that absolute growth is rewarded ($Rt_i = Rt_i+Gt_i$) and decay is penalized ($Rt_i= Rt_i-|Gt_i|$). Higher ranks represent a higher likelihood that the corresponding volume of the tenant grows aggressively. Since we only report tenants that have been potentially growing for some minimum number of events, we do not report the tenant related with a rise appearing firstly in a series.

2.3 Adaptation Decision

Design of moderation policy is not the focus of this paper, but SPIN indeed provides an open infrastructure to adopt freely policy for the moderation of service behavior to requests from aggressive tenants. It helps to set and enforce effectively proper policy to limit or isolate the negative effect of aggressive tenants on others.

3 Implementation of SPIN

In this section, the challenges met during our implementation of System Monitoring are described considering the goals of not modifying source code and providing a runtime monitoring with low performance overhead. Implementations of the other two functions focus on the algorithms in Section 2 and are omitted here to save space. System Monitor function is executed by two main phases, the instrumentation phase and the data collection phase. The instrumentation phase consists of bytecode instrumentation of the services and operations to be monitored. The data collection phase consists of running the program, gathering resource usage data, and accounting the consumption on proper tenants. They are respectively introduced in the following two sections.

3.1 Instrumentation to Maintain Accounting Stack

The Accounting Stack is implemented through the instrumentation of all service interface methods' entry and exit points with specific method calls. For every tenant, an accounting stack is built at the first time this tenant sends request for service usage. In the instrumentation phase, bytecodes of the services to be monitored is manipulated to insert the methods that maintain the Accounting Stack. On a Web service's entry a stack frame containing boundary information is pushed onto the Accounting Stack. On a Web service's exit a stack frame is popped from the stack. For every resource consumption event, resource consumed is accounted on owner of present accounting stack. In this way, a complete record is kept on the service chain accessed by this tenant and resource usage during the service time. It should be noted that we perform the instrumentation integrated with service lifecycle as a part of service deploy process. An array of statically allocated stack elements is employed here to avoid dynamic memory allocation and de-allocation during the push and pop operations. This help to address the efficiency concerns.

3.2 Data Collection by Resource Consumption Agent

Accounting Stack calculates the accounting unit. Meanwhile, determining resource consumption and billing the current accounting unit are the responsibilities of Resource Consumption Agent. We have built prototypes for two important resources, CPU and the Java heap.

The center of Resource consumption agent is a native agent with architecture presented in Figure 1. Two trackers are built to collect CPU usage and Memory usage respectively. CPU Tracker probes CPU for calculation of cycles by sampling. Every a sampling interval, the consumption of CPU in this interval are accumulated and accounted on the Current Accounting Tenant (CAT) which is reserved in Accounting

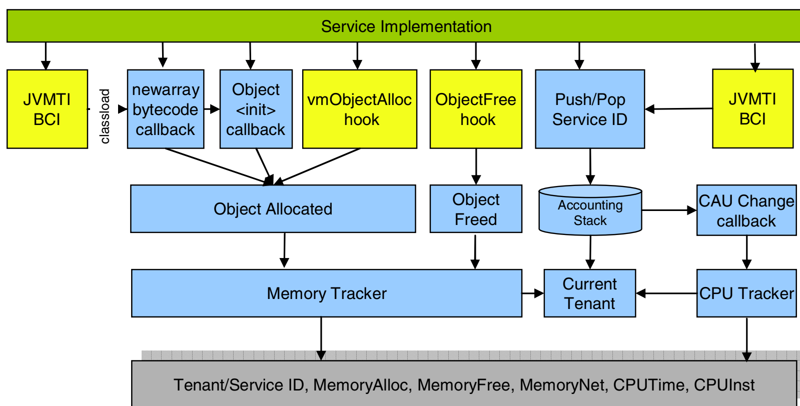

Fig. 1. Java Native Agent

Stack. Similarly, Memory tracker tracks memory allocation and de-allocation. We implement trackers by a Java Virtual Machine Tool Interface (JVMTI) agent. JVM TI [23] provides a set of standard interfaces for tracking object lifecycles and the state of JVM. A JVM TI agent can be notified of interesting occurrences through events and can control the service application through many functions, either in response to events or independent of them.

Special attention should be paid to the accounting of reusable resource, like memory allocated on Java Heap. The memory allocated will be reclaimed if it is no longer used. Each time an object is allocated, the object is also tagged with the ID of the accounting tenant. Tracking object deallocation is relatively simpler: JVM TI provides an event callback, i.e. *ObjectFree*, which notifies us every time an object is freed. On an object free event, we retrieve the tag of the freed object to determine the accounting tenant that was charged for this object. The memory consumption of that tenant is decreased accordingly.

Another point needs attention is the JNI cost caused by Accounting Stack which must communicate any change in the CAT to the Resource Consumption Agent. While our Agent runs in native space and as such the CAT must be made available in native space. To reduce the cost, we have used *java.nio.ByteBuffer*, whose instance is allocated outside the garbage-collected heap and can be accessed from both Java and native code. The use of *ByteBuffer* has been proved much more efficient to copy the CAU into native space than a Java native method does. On each object allocation, the Resource Accounting Agent retrieves the current accounting tenant from the *ByteBuffer* and charges it for the allocation.

4 Experiments and Evaluation

IBM Trade6 works as a performance benchmark and Web service sample application by providing a real-world workload, enabling performance research and verification test of the service Platform. It models an electronic stock brokerage providing Web services-based online securities trading. Routine stock operations, such as selling, watching holdings, and so on, are encapsulated into Web services and accessed by client at the runtime. We have revised Trade 6 to enable Multi-tenancy and adopt it in our experiments to SPIN's effectiveness on performance isolation and measure overhead of SPIN implementation.

Experiment Configuration: We deployed Trade6 backend Web services on a 2.66GHz desktop with 2GB RAM, Windows XP, WebSphere Application Server V6.1, and DB2 V8.2. WebSphere Application Server is configured to use 1G heap. Trade 6 provides a stress client to simulate workload and service requests and we put it on another desktop with the same configuration parameters. The two machines are connected by 100Mbps LAN.

Experiment Scenario: We simulate the case that Trade6 serves five tenants, identified as tenant 0 to tenant 4. At the beginning, requests from 5 tenants are balanced, i.e. the request numbers of different tenants are identical. After 2 minutes, the request number from tenant 0 is increased largely so that the increase has negative impact on performance of other tenant or even pushes the whole system to the edge of crashing. What we want to see from this experiment is how SPIN can help in such scenario.

Following sections give detailed experiment data and explanation on anomaly report, isolation effectiveness and performance overhead of SPIN.

4.1 Anomaly Report

The experiment concerns firstly anomaly detection of WAQ model. Two time series *alpha* (Fig.2 (a)) and *beta* (Fig.2 (b)) are got by monitoring with a sampling interval of 100 milliseconds. The alpha series and beta series are respectively the arrival process and service process of hosting system. Series *y=alpha-beta* (Fig.2 (c)) represents the changes on processing capability of service system over time. Some bursts occur around the time points 300, 600, 1000, 1600, and 2000.

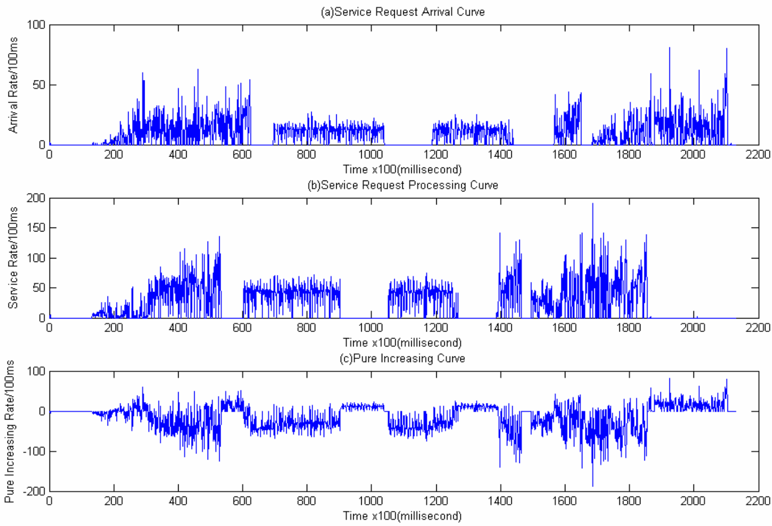

Fig. 2. Time series of service request arrival (a), request processing (c), and pure request increase (c) with settings of 2000 thread and 1500 times iterations

In our analysis, decomposition is made on the series at the resolution level one for clear experiment. WAQ threw out anomaly report after 190000 milliseconds since the start of the benchmark. In comparison, we do not adopt any moderation in the first running and find the service system finally collapse after about 20000 milliseconds later than the anomaly report.

Figure 3 presents that the predicted curve consists well with the monitored one, which demonstrates that WAQ can predict the changes both in arrival rate and service rate efficiently, no matter how wild fluctuations are. The accuracy of presented prediction (see Tab.1) is studied in terms of MRE (Mean Relative Error) [20]. Table.1 illustrates that the precision improves as the order of AR used in WAQ increases. Here we think that the parameter of order is properly set to “50” for the high precision and smaller computational complexity of WAQ.

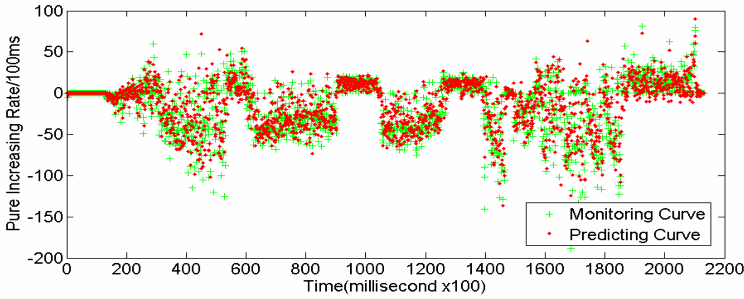

Fig. 3. Comparison of pre-series and post-series with ARP(15)

Table 1. MRE of Arrival Rate with different AR(p)

	AR(5)	AR(15)	AR(30)	AR(40)	AR(50)	AR(60)
MRE	36.94%	20.11. %	6.34%	4.68%	3.08%	2.25%

4.2 Isolation of Aggressive Tenant

After the anomaly report, the trace file written by system monitoring during service access is analyzed and resource volume of each tenant is collected. Figure 4 presents how CPU consumptions of every tenant change from the start to the anomaly report time. Other resources, such as memory, are omitted here to save space. In the forepart of the curves, each tenant consumes similar CPU cycles. A sudden rise occurs on the curve from the time of 1340. Tenant 0 is identified as the aggressive tenant because its consumptions of CPU cycles show the fastest increasing rate.

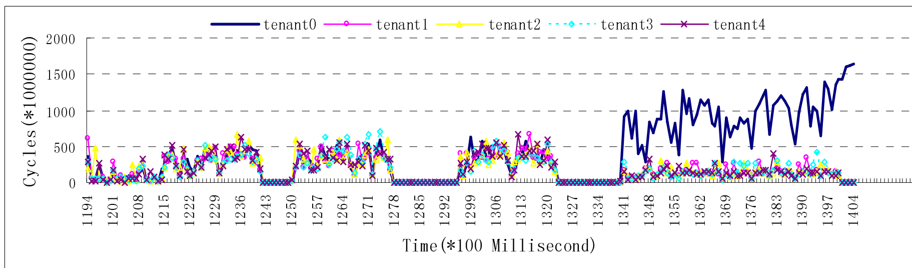

Fig. 4. CPU consumption curve for each tenant

With the rise of tenant 0, obvious drops lie on other tenants' consumption which decreases to almost 0 near the anomaly report time, 1407. Tenant 0 has already impacted negatively on other tenants. Without SPIN, the duration of this status will finally lead the non-aggressive tenants into the danger of starvation and service system into crash. The SPIN implementation adopts a direct policy to restrict and serve

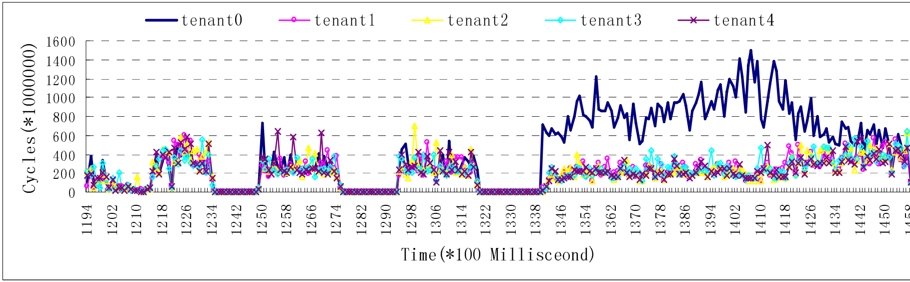

Fig. 5. CPU consumption curve with the moderation function open of SPIN

the requests from tenant 0 in the inverse ratio of its rise. Figure 5 presents the curve under the same simulation of workload with the moderation of SPIN open. The growth of Tenant 0 is limited not to reach the dangerous peak. The CPU consumptions of other tenants originally decrease and generally experience rallies. The percentages of non-aggressive tenants' consumption taking in all basically recover to the balanced state, which demonstrates the effectiveness of SPIN to keep performance isolation in Multi-tenancy environment. In practice, it is suggested to adopt a moderation policy consistent with SLA of different tenants.

4.3 Performance Overhead

To understand performance overhead brought into the original execution of service system, we watch the values of average response time before and after adoption of SPIN. To observe the values under different scales of workload, we tune two parameters, the number of threads (simulated clients) executing service access and iterations times executed in each thread. Performance overhead is calculated as $(T_i-T_o)/T_o$ for each different setting of the two parameters, where T_i and T_o are the average response time with SPIN and original benchmark respectively.

Figure 6 presents the overhead histograms with various settings of thread number (the parameter of iteration times is set to 2000) and interaction times (the parameter of thread number is set to 1500). For each setting, the benchmark runs 10 times, and the final results are obtained by calculating the geometric mean of the median of setting.

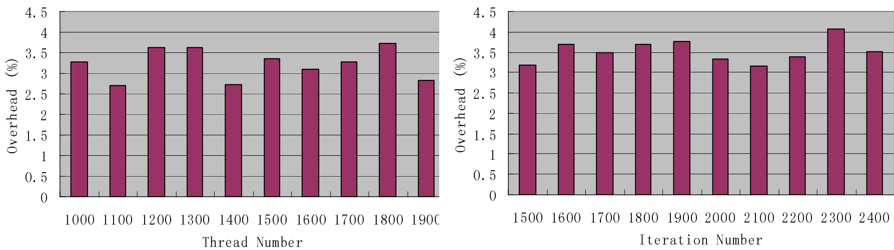

Fig. 6. Overhead histograms with various thread number and various interaction times

Overhead values with different settings are all less than 5%, which presents little negative impact caused by SPIN and its practicability under product environment.

5 Related Works

Performance isolation is not a new idea [24]. Jordan et al. [25] applies the concept of a Java resource accounting interface to isolate applications inside a JVM at the granularity of isolates to J2EE platforms. In comparison, our work focuses on performance isolation of tenants on the same copy of service. Work in [25] still depends on the resource reservation approaches and thresholds setting to limit the performance factors of isolations. Isolation for single Java virtual machines have been studied extensively [26, 27] and they focus on the security of multi-task in one JVM.

Authors in [11, 12] apply Queuing Theory in the study of Web service and adopt response time, throughout and reliability to evaluate performance of a service system. Three birth-death (BD) models are introduced in [13] to prove results on system throughput with the condition that the parameters of BD process have the same ratio. Renaud et al [14] address failure rate for the Web services market using Markov chain and Queueing Theory. They do not concentrate on the identifying how and when a service system becomes instable, but the comprehensive evaluation of performance.

In addition, there are some prior works for dealing with performance problems in server applications. That include request deletion in web servers [28], request prioritization or frame dropping in multi-media or real-time applications [29], and the creation of system level constructs supporting these application-level actions [30,31]. They share with adaptive techniques the use of runtime system monitoring and of dynamically reacting to certain monitoring events, but they differ in that focus is put on the decision of proper moderation policy. No attention is paid by them on the anomaly detection of service system. Moreover, they do not care about the resource monitoring at levels other than Process.

6 Conclusion and Future work

SPIN is purposed in this paper to achieve performance isolation of Multi-tenants on the service hosting platform with the maximum resource share. By a detection independent of any threshold, SPIN gives anomaly report in advance of the instability of service system. Resources, like CPU, consumed during service access are accounted on behalf of each tenant. Tenants whose consumption presents a trend of continuous growth are identified as aggressive ones that moderation will execute on. SPIN has been implemented open to self-tuning moderation without relying on user's input or directions. Practice of SPIN in Trade6 which has been revised to Multi-tenancy model demonstrates its accuracy of anomaly detection, effectiveness of isolation, and the low performance overhead (less than 5%).

Next step, we plan to adapt SPIN for SLA of different tenants. We will build separate anomaly detection for various tenants. Especially, proper moderation policy needs to design considering various SLA.

References

1. Carraro, G., Chong, F.: Software as a Service (SaaS): An Enterprise Perspective, Microsoft2. Corporation (October 2006), http://msdn2.microsoft.com/
2. Gianforte, G.: Multiple-Tenancy Hosted Applications: The Death and Rebirth of the Software Industry. RightNow Technologies Inc. (2005), http://www.rightnow.com
3. Chong, F., Carraro, G., Wolter, R.: Multi-Tenant Data Architecture, Microsoft Corporation (2006), http://msdn2.microsoft.com/
4. Tsai, C.-H., Ruan, Y., Sahu, S., Shaikh, A., Shin, K.G.: Virtualization Based Techniques for Enabling Multi-tenant Management Tools. DSOM, 171–182 (2007)
5. Czajkowski, G., Daynes, L.: Multitasking without compromise: a virtual machine evolution. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001 (November 2001)
6. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
7. SWSoft, Virtuozzo, http://www.sw-soft.com/virtuozzo
8. IBM. WebSphere Application Server, Trade6 benchmark, https://www14.software.ibm.com/wbapp/iwm/web/preLogin.do?source=trade6
9. Waldspurger, C.A.: Memory resource management in vmware esx server. SIGOPS Operating Systems Review 36, 181–194 (2002)
10. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Geiger: Monitoring the buffer cache in a virtual machine environment. In: The 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 14–24 (2006)
11. Hopp, W.J.: Single Server Queueing Models. In: Chhajed, D., Lowe, T. (eds.) When Intuition Fails: Insights From Basic Operations Management Models and Principles. Springer, Heidelberg (scheduled for publication, 2007)
12. Hall, R.W.: Queueing methods for services and manufacturing. Prentice-Hall, Englewood Cliffs (1990)
13. Feng, W.: Improving Service for Service Systems with Different Arriving Rate, PDCATapos. In: Proceedings of the Fourth International Conference on Volume, pp. 315–318 (August 2003)
14. Renaud, O., Starck, J.L., Murtagh, F.: Wavelet-based Forecasting of short and long memory time series [EB/OL]
15. Czajkowski, G., Eicken, T.V.: Internet Servers, Safe-Language Extensions, and Structured Resource Control. In: Proceedings of the Technology of Object-Oriented Languages and Systems, Nancy, France, pp. 295–304 (1999)
16. Hulaas, J., Kalas, D.: Monitoring of Resource Consumption in Java-based Application Servers. In: Proceedings of the 10th HP OpenView University Association Plenary Worksop (HPOVUA 2003), Geneva, Swizerland (2003)
17. Liang, S., Viswanathan, D.: Comprehensive Profiling Support in the Java Virtual Machine. In: Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 1999), San Diego, CA, pp. 229–240 (1999)
18. Sutherland, D.F., Greenhouse, A., Scherlis, W.L.: The Code of Many Colors: Relating Threads to Code and Shared State. ACM SIGSOFT Software Engineering Notes 28(1), 77–83 (2002)
19. Liu, Z.-X.: Short-term load forecasting method based on wavelet and reconstructed phase space. Machine Learning and Cybernetics 8, 4813–4817 (2005)

20. XiangXu, B., XinMing, Y., Hai, J.: Network Traffic predicting based on wavelet transform and autoregressive model. In: Tsui, F.-C., Sun, M., Li, C.-C., Sclabassi, R.J. (eds.) Recurrent neural networks and discrete wavelet transform for time series modeling and prediction, ICASSP, vol. 5(9-12), pp. 3359–3362 (May 1995)
21. Akaike, H.: Fitting autoregressive models for prediction. Annals of the Institute of Statistical Mathematics 23(1) (December 1971)
22. Mallat, S.G.: A Theory for Multiresolution Signal Decomposition: The Wavelet Representation. IEEE Transactions on pattern analysis and machine intelligence 11(7), 674–693 (1989)
23. Sun Microsystems, Inc. JVM Tool Interface (JVMTI), `http://java.sun.com/-j2se/1.5.0/docs/guide/jvmti/`
24. Barham, P., Dragovic, B., Fraser, K., et al.: Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003 (2003)
25. Jordan, M.J., Czajkowski, G., Kouklinski, K., et al.: Extending a J2EETM Server with Dynamic and Flexible Resource Management International Middleware Conference, Middleware 2004 (2004)
26. Czajkowski, G.: Application isolation in the Java Virtual Machine. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA 2000 (2000)
27. Back, G., Hsieh, W., Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In: Proceedings of the 4th International Conference on Operating System Design and Implementation (OSDI), San Diego, CA, pp. 334–346 (2000)
28. Provos, N., Lever, C.: Scalable Network I/O in Linux. In: Proceedings of the USENIX Technical Conference, FREENIX track (2000)
29. Sundaram, V., Chandra, A., Goyal, P., et al.: Application performance in the QLinux multimedia operating system. In: Proceedings of the 8th ACM International Conference on Multimedia 2000 (2000)
30. Poellabauer, C., Schwan, K., West, R., et al.: Flexible User/Kernel Communication For Real-Time Applications In Elinux. In: Proceedings of the Workshop on Real Time Operating Systems and Applications (2000)
31. West, R., Schwan, K.: Dynamic Window-Constrained Scheduling for Multimedia Applications. In: Proceedings of the IEEE International Conference on Multimedia Computing and Systems, ICMCS 1999 (1999)