MAXIMIZING PERFORMANCE OF CO-LOCATED APPLICATIONS: EXPERIENCES AND STRATEGIES WHEN CO-LOCATING MULTIPLE JAVA APPLICATIONS ON A SINGLE PLATFORM

Zhenyun Zhuang, Cuong Tran, Haricharan Ramachandra, Badri Sridharan
LinkedIn Corporation, 2029 Stierlin Court Mountain View, CA 94043, USA
{zzhuang, ctran, hramachandra, bsridharan}@linkedin.com

Abstract
Cloud (e.g., PaaS) Computing promises a cost-effective and administration-effective solution to the needs of sharing computing resources. A widely practiced cloud deployment model is to co-locate multiple applications on a single platform. While bringing efficiency to the users thanks to the shared hardware and software, the multi-tenancy characteristics also bring unique challenges to the backend platforms. In particular, the JVM mechanisms used by Java applications, coupled with OS-level features (e.g., THP), give rise to a set of problems that are not present in other deployment scenarios. In this work, we consider the problem of maximizing performance of co-located mission-critical Java applications when deploying multiple Java applications on a single platform. Based on our experiences with LinkedIn’s platforms, we identify and solve a set of problems caused by such multi-tenancy deployment. We share the lessons and knowledge we learned during the course.

// An earlier version of this paper was accepted in the conference of IEEE Cloud 2014 (Alaska, AL, USA)
// Conference paper title: "Optimizing JMS Performance for Cloud-based Application Servers"
// The differences between the conference paper and this draft are listed at the end of the draft.

Keywords: Linux performance; Multi-tenancy; THP (Transparent Huge Pages); Java performance; Cloud Computing; NUMA

1. INTRODUCTION
Cloud Computing promises a cost-effective and administration-effective solution to the traditional needs of sharing computing resources. A widely practiced cloud deployment model is to co-locate multiple applications on a single platform. While bringing efficiency to the users thanks to the shared hardware and software, the multi-tenancy characteristics also bring unique challenges to the backend cloud platforms. Cloud platforms need to deliver high performance to all the co-located applications/users, subject to SLA (Service Level Agreement). However, since multiple applications share the same computing resources, it is important to prevent different applications from adversely affecting each other. Note that depending on specific cloud solutions, users may use a mix of dedicated resources and shared resources.

Though the general problem of isolating applications and ensuring high performance belongs to the big topic of QoS (Quality of Service), as of today, there are little satisfying solutions to isolating applications while maintaining the cost-effectiveness of multi-tenancy. Note that Some solutions provide hard isolation of resources by dedicating a specific set of hardware and OS to each application (e.g., through Virtual Machines), however it incurs additional administration cost and compromises the benefits gained by resource sharing. In many of today’s multi-tenant backend platforms such as many PaaS platforms, multiple users share the same OS (and in turn the memory and cpu resources). Reasons for adopting such setup includes ease of administration and deployment. In such multi-tenancy deployments, it is critical to protect users at OS level to ensure high performance.

Different co-located applications impact each other when sharing limited computing resources including memory and cpu. When users’ consumptions of computing resources are not aligned well due to independent activities and load, the individual user’s performance may be compromised. For example, an user that suddenly aggressively consumes memories (e.g., reading files) and causes another user to swap out its pages, when the latter users resumes, its performance will be significantly penalized due to slow swap-in activities. Moreover, with the increasing popularity of multi-core and NUMA (Non-uniform memory access) ("Non-uniform", 2014) systems, additional performance overhead caused by thread migrations and context switches may cause the default application deployment model suboptimal.
Given the popularity and powerfulness of Java platforms, a significant portion of today’s backend platforms run Java. Though supporting multi-tenancy inside a single JVM has been proposed ("Java Multitenancy", 2014), it is yet to be verified and deployed. As of today, effectively supporting multi-tenancy with Java requires running multiple Java applications on the same platform. Java applications’ performance highly depend on the use of JVM and heap spaces. We have observed in our production platforms that the interaction caused by different Java applications can cause severe performance degradations. Briefly, the competitions among multiple Java applications cause inflated application pauses during certain scenarios. Moreover, we have found that some new features of Linux platforms, though intended to improve application performance, could significantly backfire, thanks to some complicated interactions between these features.

In this work, we consider the problem of maximizing the performance of mission-critical Java applications when co-locating multiple java applications on a single platform. Based on our experiences with LinkedIn’s platforms, we identify and solve a set of problems caused by co-locating multiple Java applications. In a nutshell, we consider the performance of Java applications, and identified a set of Linux OS features (e.g., THP or Transparent Huge Pages) that can adversely affect Java application performance in certain scenarios. After root-causing the problems, we provide solutions to mitigate the challenges. Though the solutions have been verified working on Linux OS, on which most of LinkedIn products run, they can easily be applied to similar setup of other platforms to achieve better performance. More importantly, the studies gained could be incorporated into future designs of JVM or OS features.

For the remainder of the paper, after providing some necessary technical background in section 2, we then motivate the problems being addressed in this paper using two scenarios in Section 3. We then present the investigations we conducted in diagnosing the performance problems in Section 4. Based on the findings, we propose the solutions in Section 5. We perform performance evaluation and show the results in Section 6. In Section 7 we highlight the learned lessons during our investigations with LinkedIn’s platform. We also present related works in Section 8. And finally in Section 9 we conclude the work.

2. BACKGROUND

We begin by providing background information regarding JVM heap management and GC (Garbage Collection), Linux memory management, and new features of THP (Transparent Huge Pages).

2.1 JVM heap management and GC

Java programs run in JVM (Java Virtual Machine), and the area of memory used by the JVM is referred to as heap. JVM heap is used for dynamic Java objects allocations and is divided into generations (e.g., Young and Old). Java objects are firstly allocated on the Young generation; when unused, they are collected by a mechanism called GC. When GC occurs, objects are checked for reference counters starting from a root object. If the reference counter of an object drops to zero, the object is deleted and the corresponding memory space is reused. Some phases of GC process require applications to stop responding to other requests, a behavior commonly referred to as STW (Stop the world) Pause. One of the important objectives of Java performance is to minimize the durations of GC pauses.

2.1 Linux memory management

Memory management is one of the critical components of Linux OS. The virtual memory space is divided into pages of fixed sizes (e.g. 4KB). Over the years, many features of memory management have been designed to increase the density and improve the performance of running processes.

Page reclaiming Linux OS maintains free page lists to serve applications’ memory request. When the free pages drop to a certain level, OS will begin to reclaim pages and add them to the free lists. When performing page reclaiming, page scanning is needed to check the liveness of allocated pages. Both background scanning (performed by kswapd daemon) and foreground scanning (performed by processes) are used. Oftentimes the foreground page reclaiming is referred to as direct reclaiming or synchronous reclaiming, which represents a more severe scenario where the system is under heavy memory pressure, and the application stops during the process.
Swapping

Swapping is designed to increase process density. When free memory is low, Linux will swap memory pages out to swap spaces to allow for new processes being invoked. When the memory space corresponding to swapped-out pages is active again, these pages will be swapped in.

THP

Transparent huge pages ("Huge pages", 2014) is a relatively new feature that aims to improve the processes’ performance. With larger page sizes (e.g., 2MB), the number of page table entries is reduced for a particular process. More virtual address translations can be covered by TLB hit ratio (Talluri et al., 1992). Though the benefit of using huge pages has long been understood, the use of huge pages was not easy. For instance, the huge pages need to be reserved when OS starts, and processes have to make explicit calls to allocate huge pages. THP, on the other hand, promises to avoid both problems and thus is enabled by default. THP allows regular pages to be collapsed into transparent huge pages (THPs) through two types of collapsing: background collapsing by khugepaged daemon and direct collapsing by applications requesting memory. Note that we use THP to denote the feature, while use THPs to denote the transparent huge pages.

Non-uniform memory access (NUMA) is a computer memory design which becomes increasingly popular on modern platforms. With NUMA, applications running on a particular CPU can gain better performance when accessing local memory compared to accessing remote memory. OS (e.g., Linux) has advanced mechanisms to optimize the allocation of memory and execution of threads. Linux also exposes tools (e.g., numacl) ("Control numa", 2014) to allow the control of NUMA policy for scheduling running processes and allocating memory.

3. Motivation

We first provide three motivating scenarios to illustrate the problems that Java applications may face in multi-tenant environments.

3.1 Experiment setup

The machine used in the experiments is an x86 64-based Intel(R) Xeon(R) X5650 SMP (Symmetric multiprocessing) NUMA 2-node machine. Each node has 6 cores, and hyper-threading is enabled. The OS version is Red Hat Linux 2.6.32-220.13.1.el6. The machine has totally 72 GB of physical memory. All default system configurations are used.

Two applications are used, one is written in Java, the other in C++. For easy presentation, we refer to the Java application as JavaApp, and the C++ one BackgroundApp. JavaApp is used to represent real production applications, while BackgroundApp simply consumes computer resources of memory to mimic production environments. JavaApp keeps allocating Java objects, and also periodically discards objects such that they can be reclaimed during JVM GC.

In real production environments, it is common for the system to experience memory pressure. To mimic such situations, BackgroundApp is designed to allocate a fixed amount of memory such that when running JavaApps they can experience certain level of memory pressure.

We consider two types of Java applications: throughput-oriented or response-time-oriented; so we will examine the application throughput as well as the response times experienced by the client requests. For throughput-oriented applications, the throughput is measured as how many Java objects are allocated per second. For response-time-oriented Java applications that serve client requests, ideally the user-experienced response time should be measured. Directly measuring the user response time is not easy, as each type of Java application has different request handling mechanisms. We instead measure the severity of STW GC pauses, since during these GC pauses, Java applications stop responding to the clients.

The JavaApp is started with varying heap sizes and with identical -Xmx and -Xms values. Other important JVM configurations are: UseConcMarkSweepGC, CMSsCavengeBeforeRemark, CMSParallelRemarkEnabled.

3.2 Scenario 1: During startup state

In this scenario, a single JavaApp is started and running. We first start the BackgroundApp to take 50 GB of memory, which leaves about 20 GB memory unused. We then start the JavaApp with 20 GB heap size.
In Figure 1(a), after JavaApp is started, it has a steady throughput of 12K/sec, which lasts for about 30 seconds. Then the throughput begins to drop sharply. The worst case sees almost zero throughput for about 20 seconds. Interestingly, after a while, the throughput comes back to steady again.

In Figure 1(b) which shows the GC pauses, we also observe similar pattern. The GC pauses initially are well below 50 milliseconds; then they jump to hundreds of milliseconds. We even see 2 pauses are larger than 1 second! After a period of about 1 minute, GC pauses drop again to below 50 milliseconds and become steady.

### 3.3 Scenario 2: During steady state

In this scenario, a JavaApp is started with 20 GB heap and enters steady state. Then a BackgroundApp starts and begins to allocate 50 GB of memory.

In Figure 2(a), we saw that the JavaApp achieves a steady 12K/sec throughput in the beginning. Then the throughput sharply drops to zero which lasts for about 2 minutes. From then on, the throughput varies wildly. Sometimes it comes back at 12K/sec, other times drops to zero again.

In Figure 2(b), the GC pauses are almost zero in steady state, then it rises to 55 seconds! From then on, the GC pauses varies a lot, but rarely come back to zero. Most of the pauses are of several seconds.

### 3.4 Scenario 3: Different deployment models

We also consider the third scenario where co-located applications are deployed with different deployment models. On NUMA machines, each node has its local set of CPU and memory. For any CPU, accessing the local memory on the same node gains better performance due to smaller access time. To facilitate the needs of finer control when deploying applications, Linux exposes a set of tools (e.g., numactl) to allow applications to bind to a particular set of CPUs and memory.

We consider two deployment models: (1) default deployment model (without CPU/Memory binding) and (2) improved deployment model (with CPU/Memory binding). We deploy two identical JavaApps on the same machine, and each JavaApp has 5GB of heap. In Figure 3 we show the averaged application throughput (over two applications) of the two models. We see that the default deployment model achieves 3.26 K/sec averaged throughput, while the improved deployment model achieves about 3.72 K/sec, a 14% difference. More importantly, from the curve, we can see that the improved (i.e., with CPU/Memory binding) deployment model has much more stable performance.

We also examined the CPU utilization of these two models and plotted the CPU idle percentage in Figure 4. We observed that the improved deployment model has about 29.8% CPU busy usage, slightly lower than the 31.9% of default deployment model, or a 7% difference. In other words, not only the default deployment has lower and unstable application throughput, but also it uses more CPU resources.
3.5 Summary

We see that in multi-tenant environments, Java applications can experience performance problems both in startup state and steady state. In addition, the default deployment model on NUMA platforms is sub-optimal. The symptoms of these performance problems as seen by users are low application throughput, increased response time and unstable performance. We will investigate the problems and root cause them in Section 4.

4. INVESTIGATIONS

We just saw that Java applications can experience significant performance degradation in multi-tenant setup. We now delve deeper into the two scenarios and identify the causes.

4.1 Investigations into Scenario-1: Startup state

In Section 3.2 we see that during the startup period JavaApp experiences undesired performance. We suspect that the undesired performance of JavaApp has to do with how JVM is started since it occurs when JVM is started. We examine the JavaApp’s RES (Resident size), which is the non-swapped physical memory a process has used. The results are shown in Figure 5(a). We see that though we specify the JVM parameters to be “-Xmx20g and -Xms20g”, JVM does not allocate the heap space from memory all at a time. Instead, the heap is allocated on the go. As more and more objects are instantiated, JVM allocates corresponding memory pages to accommodate. During the allocation process, OS checks the free page list. If the amount of free

memory is below certain level, OS will begin reclaiming pages, which takes CPU time. Depending on how severe the shortage of free memory is, the reclaiming process may significantly stall the JavaApp. In Figure 5(b) we see that free memory significantly drops to a very low level. The page reclaiming process incurs CPU overhead, as can be seen in Figure 5(c).

On Linux, when available memory is low, the kswapd daemon wakes up to free pages in the background. If the memory pressure is high, the reclaiming process will free up memory synchronously. The page reclaiming mechanism of Linux is per-zone based (Lameter, 2013), which is governed by certain zone watermarks (i.e., pages min, pages low, pages high). User-land Java applications allocate memory in NUMA nodes’ Normal zones. For our setup, there are two nodes. Node-0’s pages_high is set at 21,498 pages (about 83 MB), while Node-1’s 11,277 pages (about 44 MB). We examined the free pages of Normal zones for the two nodes based on /proc/zoneinfo, and plot the amounts in Figures 6(a,b). We found that the available pages occasionally drop below the watermarks, which could trigger direct-reclaim path. When direct-reclaim occurs, Linux freezes the applications that executing the code path, thus causing high GC pauses. In addition, direct-reclaiming typically scans many pages in order to free unused pages. In Figure 6(c) we plot the number of pages scanned by the direct-reclaiming path as reported by Linux SAR utility. We see that at the peak value, every second about 48K pages (i.e. 200 MB) amount of pages are scanned by direct-reclaiming.

4.2 Investigations into Scenario-2: Steady state
In Section 3.3 we observe that the actions of other applications can severely impact the performance of a Java application. Since the total physical memory is only about 70 GB, while the two applications together request that amount of memory, our first observation is that the system is under memory pressure. We found that there are quite a lot of swapping activities going on, as seen from the Figures 7. In Figure 7(a) we see that many memory pages are swapped out. These swapped-out pages belong to the JavaApp. Later on in Figure 7(b), many pages are then swapped in. We can see from Figure 7(c) that the taken swap space is up to 7 GB. During these time, if JavaApp experiences GC activities, and GC needs to scan objects to collect dead ones. If the scanned objects are allocated on the pages that are swapped out, they need to be brought back to memory from swap space first. Swapping-in takes time, as the swap space is typically on disk drives. Thus, JavaApp see high GC pauses.

Though swapping activities affect GC pauses, we suspect that they alone are unable to explain the excessive pauses we see (i.e. 55 seconds). Our suspicion is justified by our examination of the GC pauses, many of which show high sys time. In Figure 8(a) We observed that the system is also under severe cpu pressure. The high cpu usage cannot be entirely attributed to swapping activities as swapping typically is not cpu intensive. There must be other activities contributing to the cpu usage. We examined various system performance statistics, and identified a particular Linux mechanism of THP that significantly exacerbates the performance degradation.

With THP enabled, when Linux applications allocate memory, preference is given to allocations of transparent huge pages of 2 MB size rather than the regular pages of 4 KB. We can verify the allocation of transparent huge pages in Figure 8(b), which shows the instantaneous number of anonymous transparent huge pages. Since THPs are only allocated to anonymous memory, the figure practically shows the total THPs of the system. At the peak we see about 34K THPs, or about 68 GB.

We also observed that the number of THPs begins to drop after a while. This is because some of the THPs are split in response to low available memory. When system is under memory pressure, it will split the THPs into regular pages to be prepared for swapping. The reason for splitting is because currently Linux only supports swapping regular pages. The number of splitting activities are seen in Figure 8(c). We see that during the five minutes, about 5K THPs are split, which corresponds to 10 GB of memory.

At the same time, Linux attempts to collapse regular pages into THPs, which requires page scanning and consumes cpu. There are two ways to collapse: background collapsing and direct collapsing. Background collapsing is performed by khugepaged daemon. We occasionally observed that the khugepaged daemon tops cpu usage. Direct collapsing by applications trapping into kernel and allocating huge pages has even more severe consequences in terms of performance penalty, which can be seen by direct page scanning count.

An even more troublesome scenario THP can ran into is that the two contradicting activities of compacting and splitting are performed back and forth. When system is under memory pressure, THPs are split into regular pages, while a short while later regular pages are compacted into THPs, and so on and forth. We have observed such behaviors severely hurting application performance in our production systems.

4.3 Investigations into Scenario-3: Different deployment model

In Section 3.4 we observe that the default deployment model does not result in optimal performance of co-located applications on NUMA platforms. The application performance can be improved by enforcing memory and cpu binding. These results highlight the necessity of optimizing the deployment model when co-locating multiple applications.

Facing multiple nodes, OS running on NUMA needs to make two critical decisions: (1) decide which memory needs to be allocated when applications request memory; (2)
decide which CPU should execute a running task (i.e., thread). Though Linux, as well as other modern OS, employ sophisticated algorithms to allocate memory when applications request memory and to assign threads to CPUs when scheduling running tasks, the algorithms oftentimes do not result in optimal performance. OS typically needs to strike the balance among multiple performance factors including CPU load balancing to provide a generic mechanism accommodating all sorts of scenarios. Moreover, enterprise Java applications typically feature more than hundreds of threads, which make the CPU scheduling even harder. Because of this, the memory allocations and threads scheduling may not be optimal for a particular deployment scenario. For instance, a Java thread that is dynamically scheduled to run on a different CPU based on CPU load balancing considerations may have to frequently access remote memory, resulting sub-optimal performance. In addition, migrating among different computing nodes also risk losing the benefits of warm CPU/TLB caches.

On the other hand, for typical business deployment scenarios, the deployment needs are fixed in the sense of: (1) how many applications will be deployed; (2) the memory footprint and CPU usage of these applications; (3) how many computing nodes on the NUMA platform; (4) how much memory and how many CPUs on a computing node, etc. By considering these fixed parameters, a much optimized deployment model can be realized.

4.4 Summary

We saw that JVM mechanisms, coupled with OS-level features, give rise to unique problems that are not present in other deployment scenarios. Though the investigations and findings are Linux specific, given the wide deployment of Linux OS, particularly in server markets, the findings apply to a significant portion of multi-tenant backend platforms. In addition, other platforms are expected to expose similar problems of varying degree of severity. On one hand, JVM mechanisms are largely universal across OS platforms. On the other hand, most OS platforms have mechanisms of swapping and reclaiming. These similarities, along with the identical requirements with regard to multi-tenancy in cloud environments, makes us believe our findings can help on similar problems and solutions in other cloud platforms.

Note that to help illustrate the problem and understand the causes, we consider the particular motivation scenarios where the memory requirement almost exactly matches the physical memory. However, based on our production experience, these scenarios do occur frequently in real productive environment. This is because the fact that deployment often times over-commits applications to allow resource sharing and reduce cost.

5. Solution

We now present solutions to prevent Java applications running with multi-tenancy from performance degradation.

5.1 Overview

Our solution consists of a set of four design elements, each targets a specific aspect of the problems discussed in Section 4. Applying any individual element will help to some extent, however, all the four design elements need to be applied and work together to get the full benefit.

- **Design Element I: Pre-allocating JVM heap space.** We know that JVM heap space is not allocated until they are absolutely used. When new heap space is needed to accommodate new objects allocation requests. Linux OS needs to allocate memory pages for the heap growth, which may trigger heavy page reclaiming and hurt performance. The design element pre-allocates all the heap space such that avoiding on-the-fly page allocation by Linux OS. To enforce heap pre-allocation, Java applications need to be started with “-XX:+AlwaysPreTouch”. The side-effect of this design is the increased time taken for the JVM to start, which needs to be taken into account when applying the design element.

- **Design Element II: Protecting JVM heap spaces from being swapped out.** We learned that when GC occurs, the corresponding memory pages need to be scanned. If these pages were swapped out, they need to be swapped in first, which incurs delay and hence GC pauses. The design element protects the pages from being swapped out. Linux OS allows turning off swapping, however it applies to all applications and all memory spaces. An ideal implementation is to allow fine tuning of swapping in terms of which applications and what memory areas. For instance, the use of cgroup (“Linux cgroups”, 2014) to finely control which applications to swap. However, the complexity and administration cost may make such mechanisms a over-killer. On the other hand, we realize that for most multi-tenant platforms as in LinkedIn, these platforms are dedicated to only running homogenous Java applications. Note that Homogenous applications are identical applications running same code base except serving different customers. In these scenarios, it justifies to simply turn off swapping for applications.

At this time, Linux has two approaches to turn off swapping: (1) by issuing “swapoff ” command and (2) by setting swappiness=0. The difference between them is that the former completely forbids any swapping activity while the latter only discourages swapping. When the system is unable to handle new process’s memory request, the former approach will kill another process to free memory, while the latter approach begins swapping anyway. So depending on deployment scenarios and requirements, they need to be chosen carefully.

http://hipore.com/ijcc
Design Element III: Dynamically tuning THP. Though we have seen that enabling THP feature could cause critical performance penalty, THP provides performance gains in other scenarios. The bottom line is to enable THP when it can bring benefits, while disable it when it could cause troubles. The first observation we make is that THP exposes performance penalty mostly when the system’s available memory is low. When that happens, existing THPs need to be split into regular pages for swapping out. Thus, it is better to disable THP entirely when the system is under memory pressure. Since Java applications allocate heap when started (particularly when started with "+XX:+AlwaysPreTouch"), it is important to decide on whether to allocate THPs to a Java application when started. Thus, we choose to use the memory footprint size of the Java application as the memory threshold to decide whether to turn on or off THP. When the available memory is significantly larger than the application’s memory footprint size, then THP is enabled, as the system is unlikely under memory pressure after launching the particular application. Otherwise, THP is disabled. Since many dedicated backend platforms like LinkedIn’s are hosting homogenous applications, assigning applications’ footprint size is a simple while effective decision.

Moreover, regular pages need to be collapsed into THPs before the huge pages can be allocated to applications. Thus, part of the element is to decide when to allow THP collapsing. We propose to base the decision on the direct page scanning rate and the cpu usage of khugepaged. In summary, the design element consists of 3 components which control different collapsing types separately: (1) When available memory drops below the threshold, disable THP; (2) When direct page scanning is high, disable direct THP collapsing; and (3) when khugepaged daemon appears to be a cpu hogger, disable background THP collapsing.

Design Element IV: Explicitly hardware binding. When the deployment requirements (e.g., number of applications, memory/CPU footprint, characteristics of NUMA platforms) are known, this design element binds the hardware resources (i.e., CPU and memory) to the applications to encourage local memory access and warm CPU/TLB cache. With the presence of multiple computing nodes (each with local CPUs and memory) on a NUMA platform, this design element will categorize co-located applications into a number of sets, each set is bound to one or more computing nodes. Note that when an application’s memory or CPU footprint exceeds that of a single computing node, multiple computing nodes can be combined to form a group. For any binding between (one or more) applications and (one or more) computing nodes, the following targets are required: (1) the aggregated memory footprint of the bound

http://hipore.com/ijcc
applications should not exceed the total memory of the computing nodes; (2) the aggregated CPU footprint of the bound applications should not exceed the total CPU resources of the computing nodes. For example, assuming a 2-node NUMA platform, and each node has 6 CPUs and 48GB of memory. Given 3 applications with memory/CPU footprints of (4-CPU, 20GB), (2-CPU, 20GB) and (5-CPU, 40GB), the first two applications will be bound to one computing node, and the third application will be bound to the other computing node.

5.1 Algorithm

We classify the designs into a set of 3 algorithms as following. All the Java applications that will be run on a particular Linux system is firstly deployed with explicit hardware binding. This problem is similar to the traditional knapsack problem, which is NP-complete. We propose a greedy algorithm to expedite the processing, as shown in Figure 9 Algorithm 1. For easy description, we assume no application has larger CPU/memory footprint than a single computing node. Specifically, the greedy algorithm firstly initialize empty “knapsack” for each computing node (totally $\text{NumNode}$ nodes), denoted by $S_k$. Then it ranks all applications based on CPU footprint, forming a list of $\text{List}_{\text{app}}$.

The sorted applications then will be moved to the knapsacks one by one, starting from the application with largest CPU footprint. For every movement, the aggregated memory footprint for the current knapsack is compared against the memory resource provided by a computing node. If the former exceeds the latter, then the current application is skipped; otherwise the application will be moved to the knapsack. Every time a movement is completed, the algorithm will goes to the next knapsack. Once all knapsacks are checked, for the purpose of balancing the CPU usage among computing nodes, the ordering of applications based on CPU footprint is reversed. Then the previously described movement process will start until $\text{List}_{\text{app}}$ is empty.

For each computing node (or set of computing nodes), all the Java applications are characterized into three sets based on two types of requirements: (1) strict low pause requirements and (2) short startup delay. Specifically, as shown in Figure 9 Algorithm 2, $S_{\text{lowpause}}$ contains the Java applications that require low pauses, $S_{\text{lowstart}}$ contains those can tolerate slow startup, while $S_{\text{others}}$ are others. Note that the first two sets may have common elements.

The algorithm then adjusts the swapping configurations of the system. If all the Java applications need to be protected from swapping, then it simply adjusts the system swapping configuration, as described in Figure 9 Algorithm 2. Otherwise, individual Java applications can be separately adjusted with regard to swapping.

Before any Java applications is started, the algorithm applies “-XX:+AlwaysPreTouch” to the Java applications that belong to the intersection of $S_{\text{lowpause}}$ and $S_{\text{lowstart}}$. In other words, only Java applications that need to guarantee low GC pauses and can tolerate slow startup are started with pre-allocating heap spaces.

After Java applications are started, every $T$ time (adjustment period), the algorithm finely tunes THP. Firstly, the algorithm obtains current system performance statistics including free memory size. It then decides to enable or disable THP, as shown in Figure 9 Algorithm 3. Note that once THP is turned off, the algorithm simply skips the following steps as finer knobs are disabled inside THP.

The algorithm then checks whether it should turn on or off the background and direct THP collapsing independently. For direct THP collapsing, it relies on the past period’s statistics of direct page scanning. If it appears to be having heavy direct page scanning activities, it will turn off the knob. Otherwise, direct THP collapsing is enabled. Similarly, for background THP collapsing, it relies on the
activities of \textit{khugepaged} as shown in cpu monitoring utilities such as \textit{top}. If \textit{khugepaged} appears to be hogging cpu, that knob is turned off. Otherwise, the knob is turned on. The tuning of THP is shown in Figure 9 Algorithm 3.

6. Evaluation

We now present the evaluation results of all four design elements. We use the same JavaApp as described in Section 3, and also use the BacgroundApp for creating deployment scenarios with different available memory sizes.

<table>
<thead>
<tr>
<th>Heap size (GB)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start delay (Sec)</td>
<td>0.5</td>
<td>0.8</td>
<td>1.6</td>
<td>2.5</td>
<td>7.3</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

6.1 Pre-allocating JVM heap space

We begin by evaluating the design element of pre-allocating JVM heap space. We let the BackgroundApp to take 50GB of memory, and then start the JavaApp with 20GB heap. Recall that the machine has 72 GB of memory, the above setup creates a scenario with memory pressure. For comparison we start the JavaApp in two scenarios: without and with pre-allocating heap. As shown in Figure 10(a,b), without heap pre-allocation, the application throughput sharply drops to almost zero after about 30 seconds, which is correlated with significant GC pauses. In the scenario with pre-allocated heap, as shown in Figure 10(c), the application performance is very stable.

Pre-allocating JVM heap space does increase the startup latencies of Java applications. Applications begin to respond only after all the heap space is allocated. We quantify the startup latencies of a JavaApp of different heap sizes and found that it roughly takes about 4 seconds for every 10GB of heap size, as shown in Table 1.

6.2 Protecting JVM heap spaces from being swapped out

To evaluate the effectiveness of protecting JVM heap space from swapping, we firstly run the JavaApp with pre-allocated heap of 20GB for 150 seconds. We then start the BackgroundApp to take 50GB of memory, which encourages page swapping. In the scenario where JVM heap space is not protected (i.e., by setting swappiness to 100), we see that after about 3 minutes, the JavaApp’s performance drops sharply, as shown in Figure 11(a). In Figure 11(b), we confirmed that some heap memory is swapped out. In the scenario where JVM heap is protected, we observe much better performance in Figure 11(c). Note that the lower performance after 3 minutes are caused by THP activities.

6.3 Dynamically tuning THP

We then evaluate the design element of dynamically tuning THP. We consider the scenario where JavaApps are started with abundant available memory and less-abundant available memory. Specifically, the first JavaApp is started without other applications running. It then runs for 3 minutes and stops. After that, a BackgroundApp takes 45GB of memory, then another JavaApp is started and runs for 3 minutes. Note that for these runs, the first two design elements are both enabled.

For the above scenario, 3 mechanisms are considered: THP is turned off, THP is turned on, and THP is dynamically tuned. The adjustment period is set to be 2 seconds. The results are shown in Table 2. We see that when THP is off, JavaApp-I achieves the lowest throughput of 12 K/s, while the other two mechanisms have THP enabled and hence see higher throughput of 15 K/s.
TABLE 2: JAVAAPP THROUGHPUT (K ALLOC/SEC)

<table>
<thead>
<tr>
<th>Mechanisms</th>
<th>THP OFF</th>
<th>THP ON</th>
<th>Dynamic THP</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaApp I</td>
<td>12</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>JavaApp II</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

For JavaApp-II, since it is running under memory pressure, turning THP off gives the highest throughput. Dynamically tuning THP results in less throughput (12K/s) than turning-off THP, but outperforms THP-on since it turns off THP when necessary. Note that dynamically tuning THP brings the benefit of accommodating more scenarios, particularly in scenarios where the system usage is unpredictable thus manual configuration of THP is not desirable.

We also notice that for JavaApp II, the performance benefit brought by Dynamic THP when compared to THP-ON is not significant (i.e. 1K/s), that is because of the relatively simple scenario we considered and hence less performance improvement. However, in other scenarios such as those illustrated in Section 3, as well as in real productive environments, we have observed much more significant improvement by turning off THP appropriately. Thus, we believe the algorithm used by Dynamic THP needs to be finely tuned, which will be our future work.

6.4 Explicit hardware biding

As we described in Section 3, the NUMA system has 2 computing nodes, and each node has about 36GB memory and 12 CPUs. We consider 3 applications with the following memory/CPU footprints. App-A and App-B each uses 5GB memory and about 3 CPU; they also contain a single working thread. App-C contains 2 concurrent working threads, and the CPU footprint is about 6 CPUs using 10GB memory.

TABLE 3: JAVAAPP THROUGHPUT (K ALLOCS/SEC)

<table>
<thead>
<tr>
<th>Application</th>
<th>App-A</th>
<th>App-B</th>
<th>App-C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/O binding</td>
<td>2.57</td>
<td>2.72</td>
<td>4.52</td>
<td>9.81</td>
</tr>
<tr>
<td>W/  binding</td>
<td>2.95</td>
<td>2.93</td>
<td>5.38</td>
<td>11.26</td>
</tr>
<tr>
<td>Improvement</td>
<td>14.8%</td>
<td>7.7%</td>
<td>19.0%</td>
<td>14.8%</td>
</tr>
</tbody>
</table>

Following the greedy algorithm we presented in Section 5, App-C is bound to the first computing node, while App-A and App-B are bound to the other computing node. In the following, we will show the performance results of all applications, as well as the CPU usage of the entire platform.

Figure 12 shows the application throughput without explicit hardware binding, while Figure 13 shows the case with explicit hardware binding. We also display the average values in Table 3. We see that for all applications, explicit hardware binding achieves significant performance gains. Overall, the gain is about 15%. Probably more importantly, the performance of all applications are much more stable with explicit hardware binding, as displayed in the figures.

We also show the cpu usage comparisons in Figure 14 and observe that with-binding actually uses less CPU resources (i.e., 47% vs. 46.2%).

7. LESSONS LEARNED

During the investigations into the performance problems experienced with our production platforms, we have learned several lessons which are summarized in this section.

Multi-tenant Java platforms expose extra performance challenges. Multi-tenant platforms expose unique challenges with regard to the tradeoff between cost and performance. On one hand, the goal of serving multiple tenants on a single system is to save cost by encouraging resource sharing; on the other hand, independent activities of tenants may affect each other. The interactions between tenants oftentimes are further complicated by the OS features, particularly the memory management. When the tenant applications are running in JVM, we also need to consider the JVM and GC mechanisms due to the fact that Java maintains its own heap. We have seen in this work that the Java heap is not entirely allocated in the beginning. Though the design has its own benefits, it could cause performance problems in certain scenarios.

Be cautious about Linux’s new features (optimizations). Linux has been constantly incorporating new features in the hope of optimizing the systems and improve performance. Most of the mechanisms are successful over the years, however performance engineers should never be careful enough about these features. In this paper, we have shown that the new feature of THP proves to cause significant
performance problem when the system is under memory pressure. We have to admit that the idea of THP is fantastic, as it significantly eases the way huge pages are used. The pitfalls of THP we have seen are mainly due to the less-mature implementations of the idea. We believe as its implementations mature, THP will be able to bring more performance benefits in more circumstances. However as a general guideline, we need to deeply understand the internal mechanisms of any new OS features as well as the scenarios where a feature works or does not work. Root causes can come from seemingly insignificant information. Linux OS emits significant amount of system statistics (e.g., /proc), and most of us mostly only examine a small subset of these statistics. Over years of performance investigation lives, we have encountered many scenarios where commonly examined statistics shed little light on the performance problems. For instance, in this work, we have seen heavy direct-reclaim-caused page scanning activities, however free memory is sufficient. After we turn our eyes to examining per-zone statistics emitted by /proc/zoninfo, we nailed down the zones that are desperate for free memory and hence triggers direct page scanning.

8. Related Works

Multi-tenant cloud platform Cloud Computing model is being increasingly deployed as a fast and economic solution to serving multiple users. Despite the distinctions among the three commonly accepted service models (i.e., IaaS, PaaS, and SaaS), any large scale cloud computing environment requires the instantiation of multiple tenant applications (Woolen, 2010) or VMs (Virtual machines) (Bugnion et al., 2012). (“Oracle virtualization”, 2014) on a single hardware and/or software. Though certain techniques such as Logical Domains (“Oracle virtualization”, 2014) can isolate resources dedicated to individual tenants, to achieve highest level of resource sharing and hence cost saving, system resources have to be shared by multiple tenant applications. Such resource sharing, particular memory-sharing, gives rise to certain unique challenges.

Java performance for server applications Bearing many advantages including platform-independence, Java is used as one of the top languages to build and run server applications. Ever since its birth, extensive works have been done to finely improve its performance in various deployment scenarios (Hunt and John, 2011), (Taboada et al., 2013). (“Usage share”, 2014). However, most of the performance studies focused on the Java/JVM itself. Our work, on the other hand, deals with an undesirable deployment scenario caused by system level resource shortages and multitenant interactions.

Linux memory management and system optimizations Linux has long been the mostly deployed OS in server market (“Usage share”, 2014). Its key component of memory management has seen a plethora of advanced features being designed over the past years (Bovet and Cesati, 2005). To accommodate the fact of limited RAM and requirement of supporting multiple processes, paging and swapping are continuously optimized to improve the system/application performance (“Huge pages”, 2014), (Cervera et al., 1999), (Oikawa, 2013). Meanwhile, to better fit a particular setup, various system and configuration optimizations are researched extensively (Johnson et al., 2005).

9. Conclusion

We studied the challenges faced in a typical multi-tenant cloud platforms with multiple co-located Java applications. We identified that JVM mechanisms used by Java applications, coupled with OS-level features, give rise to a set of problems that are not present in other deployment scenarios. We propose a solution suite that addresses these problems and shared the experiences we learned.

10. References

“Control numa policy for processes or shared memory,” http://linux.die.net/man/8/nnumactl, 2014


Authors

Zhenyun Zhuang received his Ph.D. and M.S. degrees in Computer Science from Georgia Institute of Technology. He also received his M.S. degree in Computer Science from the Tsinghua University and B.E. degree from in Information Engineering from Beijing University of Posts and Telecommunications. He has been actively conducting research in various areas including system and Java performance, wireless communications, mobile networks, distributed systems, cloud computing and middleware platforms.

Cuong Tran currently is a staff engineer at LinkedIn. He worked at Yahoo and Symantec as a system architect. He also worked as an architect at Veritas Software and director of technology at Adforce. With 10+ patents/papers/blogs, he is particular interested in the areas of system/application performance and web/mobile performance, system performance and performance monitoring/analysis tools.

Badri Sridharan currently is a director of engineering at LinkedIn in LinkedIn. He received his MS in Computer Engineering from University of Wisconsin-Madison, and BE in Electronics and Communication from B.M.S College of Engineering.

Leading the performance division, his interest covers all areas of system/application performance.

Differences between the conference paper (IEEE Cloud 2014) and this draft

1. Major extensions:

- **Motivation**
  Added an entirely new sub-section of 3.4 (Scenario 3: Different deployment models) and 4 figures

- **Investigations**
  Added an entirely new sub-section of 4.3 (Investigations into Scenario-3: Different deployment model)

- **Solution**
  Added an entirely new design element of "Explicitly hardware binding” Added a new algorithm (Algorithm 1)

- **Evaluation**
  Added an entirely new subsection of 6.4 (Explicit hardware binding) and 8 figures

2. Minor extensions/changes

- **Title:**
  Changed the title to more properly cover all the contents.

- **Abstract**
  Modified the abstract correspondingly to accommodate the added contents.

- **Introduction**
  Added the new contents and rephrased several paragraphs.

- **Background**
  Added the knowledge of NUMA and tools to control NUMA policy.

- **References:**
  - Added NUMA reference.
  - Added numactl reference.