Efficient Application Placement in a Dynamic Hosting Platform[*]

Zakaria Al-Qudah
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
zma@case.edu

Hussein Alzoubi
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
hfa1@case.edu

Mark Allman
International Computer Science Institute
Berkeley, CA 94704 USA
mallman@icir.org

Michael Rabinovich
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
misha@eecs.case.edu

Vincenzo Liberatore
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
vl@case.edu

ABSTRACT

Web hosting providers are increasingly looking into dynamic hosting to reduce costs and improve the performance of their platforms. Instead of provisioning fixed resources to each customer, dynamic hosting maintains a variable number of application instances to satisfy current demand. While existing research in this area has mostly focused on the algorithms that decide on the number and location of application instances, we address the problem of efficient enactment of these decisions once they are made. We propose a new approach to application placement and experimentally show that it dramatically reduces the cost of application placement, which in turn improves the end-to-end agility of the hosting platform in reacting to demand changes.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed systems—*Distributed applications, Network operating systems*

General Terms

Performance , Design

Keywords

Application servers, Startup performance, Dynamic placement, Web hosting

1. INTRODUCTION

Web hosting has become a $4B industry [27] and a crucial part of Web infrastructure. Over half of small and mid-size businesses utilize hosting to implement Web presence [14], and 5,000 Web hosting companies with over 100 clients were reported in 2006 [6]. While the current prevalent practice involves allocating (or providing facilities for) fixed amounts of resources to customers, the fiercely competitive industry landscape has lead to a growing interest in dynamic hosting. In this approach, a hosting platform maintains a shared pool of resources and reassigns these resources among applications as dictated by the demand. By reusing the same resources to absorb demand spikes for different applications at different time, dynamic hosting promises better resource utilization and lower cost through the economy of scale.

A typical approach to implement dynamic hosting involves dynamic placement of a variable number of application instances on physical servers, and most research work in this area has been focusing on the algorithms to decide on the number and location of the application instances (e.g., [11, 19, 20, 12, 25, 26]). We address an orthogonal question of an efficient way to *enact* these decisions once they are made. For example, in the case of a flash crowd or an application-level denial of service attack, if the platform decides to add more resources to an overloaded application, we would like this decision to be enacted as fast as possible.

Currently, application placement takes in the order of minutes and, as we show, is resource intensive. This compels placement algorithms to try to minimize the rate of application placements to reduce their effects [25]. Our approach reduces dramatically these costs, and by doing so, it will enable a new class of more *agile* application placement algorithms than the existing algorithms handicapped by the current heavy-weight application switching.

Our approach includes three key components. First, instead of starting and stopping applications on demand, our scheme pre-starts an instance of *every* application on *every* available machine in a hosting platform: the applications that are intended to handle users' requests are active and the rest stay suspended on disk in a "standby" mode. Enactments of application placement decisions using this approach translate to activating suspended applications and suspending active applications, which is much more efficient in our targeted environment, where an application operates within an application server (AppServ).

Second, we observed that memory paging, which modern

[*]This work is supported in part by the NSF under Grants CNS-0615190, ITR/ANI-0205519, and CNS-0831535.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

operating systems use to move applications' state between disk and main memory, becomes the bottleneck in application placement. We resurrect long-abandoned process swapping (moving an entire process between disk and memory [5]) to address this problem. We demonstrate how process swapping avoids the inefficiencies of modern OS memory paging techniques when activating a suspended AppServ.

Finally, we recognize that modern operating systems replaced swapping with paging for a good reason, which is that paging is more efficient in fine-grained scheduling among active tasks. Thus, rather than replace paging with swapping, we combine the benefits of both memory management techniques: we allow the OS to continue using normal paging for memory management of active tasks, and use swapping only to process application placement events, which are incomparably less frequent than scheduling events among concurrent tasks. To this end, we introduce two new OS functions, allowing the enacting agent, which we refer to as *local controller*, to explicitly swap in or out a suspended AppServ, while leaving intact normal paging for active tasks.

We have built a prototype of a hosting platform that implements the above ideas, including the changes to the Linux kernel of the hosting servers to provide the above OS functions. We show that our approach offers a dramatic speed-up of application placement enactment, investigate the factors contributing to this improvement, and demonstrate its positive end-to-end effect on the hosting platform agility.

2. ARCHITECTURE OVERVIEW

We consider a hosting platform that runs typical three-tier applications, which include a Web tier, an application tier, and the back-end database tier. The web and application tiers are often combined into a single tier implemented within an application server such as JBoss Application Server (or JBoss for short) and we follow this arrangement in our experiments. However, our approach is fully applicable to the full three-tier arrangements as well. Different applications exhibit performance bottlenecks at different tiers. We focus on the application tier; see [22, 15] for examples of the work addressing the important issue of the back-end tier performance.

The components in our hosting architecture essential for the discussion in this paper are shown in Figure 1. The central controller decides how many instances of various applications should run on which servers, as well as how incoming requests should be routed among application instances. The central controller communicates its routing decisions to the request router and application placement decisions to the affected server machines. The central controller bases its decisions on reports from measurement modules, which monitor the load on every server.

The request router directs incoming requests from external clients on the Internet to a server machine that currently hosts a replica of the requested application. The request router can be a load balancing switch (e.g., [8]), a request-routing DNS server (e.g., [1, 4]), or an HTTP gateway (e.g., [9]). The request router applies the policies and configurations that the central controller issues at the hosting platform level. A local controller runs on every server machine and enacts application placement decisions from the central controller via *Start(app)* and *Stop(app)* messages.

Shared hosting platforms differ in the way they allocate resources to applications. Three main alternatives include (i) allocating entire physical machines to applications, so that resource allocation involves switching a machine from one application to another; (ii) allowing a physical machine to be shared among different applications at the level of OS processes; and (iii) running each application instance inside its own virtual environment, either a virtual machine or a container, and and sharing physical machines among multiple virtual environments. Our current approach addresses the first two ways of resource allocation while focusing more on the second approach since it is more general. In particular, we assume that application placement occurs at the granularity of the application servers running the applications in question, and we equate applications with their application servers unless it may lead to a confusion. We further assume that AppServ instances are started on the machine with operating systems already running. If different applications require different operating systems, the platform would manage pools of machines running each OS type separately.

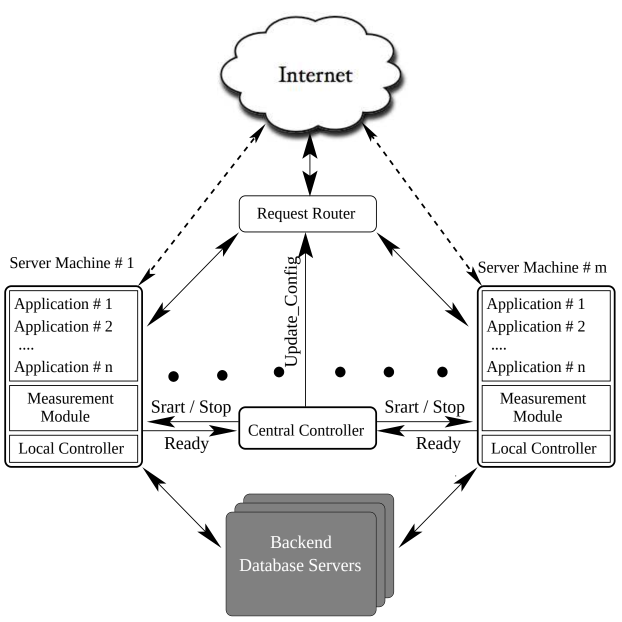


Figure 1: General architecture of a hosting platform.

The virtualized resource sharing provides a higher degree of isolation between applications at the expense of performance overhead. With virtual machines, this overhead can be significant depending on the application's operation mix [17]. In a parallel study, we observed the problem of high-cost application placement in the virtual machine environment as well [18]. We believe our techniques should also be useful in virtualized environments and plan to explore them in this context in the future.

Finally, while our testbed mimics a single data center, our techniques and findings are directly applicable to geographically distributed hosting platforms, since they face exactly the same issues in enacting a given application placement decision on a given server machine.

3. PLACEMENT MECHANISMS

This section describes a number of approaches for application placement. We begin with a straightforward optimization of a currently common technique and then proceed with several alternatives we explore in this paper.

Regular Startup. Dynamic application placement mechanisms often deploy AppServ instances from scratch, including copying the necessary files onto the target servers [21]. An obvious optimization of this scheme (which is sometimes approximated within a data center by using a shared file server) is to pre-deploy all applications everywhere, and simply start and stop AppServ instances as needed. We refer to this application placement mechanism (including the pre-deployment) as *regular startup*. As we will show later, even with pre-deployment, regular startup incurs high startup time and resource consumption. This motivates our search for more efficient application placement techniques.

Run-Everywhere. A straightforward alternative to regular startup is to run an instance of every application on every available server machine and then simply direct requests to application instances designated to be active. The intuition is that unused instances will remain idle, allowing the OS to prioritize resources to the active AppServ instances. We refer to this technique as a *run-everywhere* approach.

Suspend/Resume. In the run-everywhere approach, the operating system on each server makes resource management decisions without any knowledge about which AppServs are intended to be active. Unfortunately, as we show in Section 4, idle AppServ processes defeat the OS's general resource management by performing regular housekeeping that makes them appear active to the OS. In other words, even though the *platform* considers an AppServ idle, the AppServ's background tasks make it active in the OS's eyes. We synchronize these disparate views by allowing the platform to explicitly indicate which AppServs should be active. Specifically, the local controller responds to the *Start()* and *Stop()* commands by issuing SIGCONT and SIGSTOP signals (which are not masked by JBoss) respectively to the designated process. We denote this technique *suspend/resume.*

Enhanced Suspend/Resume. We attempt to enhance the agility of suspend/resume by addressing the following inefficiencies. First, when paging-in the memory pages of a resumed process, the operating system brings them into main memory *on-demand* (i.e., only when the resumed process generates a page fault). Thus, the resumed process alternates between CPU-bound and I/O-bound states. To reduce scheduling overhead our first enhancement prefetches the memory pages of the process to be resumed in bulk *before* waking it up with the SIGCONT signal.

Second, a resumed process will be delayed if the operating system needs to free memory to accommodate the resumed AppServ. To avoid this delay, our second enhancement attempts to maintain an amount of free memory sufficient for an average AppServ instance. Furthermore, we free this memory by *pre-purging* entire suspended AppServ from memory to disk. The rationale for pre-purging the entire AppServ is two-fold. First, application placement changes represent coarse-grained decisions. Thus, we know a recently suspended AppServ will not be activated for a long (in computing terms) time, and will likely be paged out eventually anyway. Second, bulk pre-purging is likely to place all pages of the AppServ on disk close to each other. As a result, when this AppServ is to be activated again, the activation should be faster because of fewer disk head movements. We denote the approach incorporating prefetching and pre-purging *enhanced suspend/resume*.

In short, the enhanced suspend/resume involves moving the entire AppServ process between main memory and disk. Interestingly, this mechanism, known as swapping, pre-dates the current paging (moving data between main memory and disk at the granularity of a *memory page*) [5]. Process swapping has been replaced by paging in general purpose operating systems to allow fine-grained scheduling of concurrent tasks. However, in the hosting environment, application placement events are incomparably less frequent than scheduling events among concurrent tasks. Thus, we revisit the merits of swapping in the operating systems of hosting servers to support application placement. It is, however, important that the efficiency gains of paging over swapping are not lost in the general operation of the hosting machine.

Our approach combines the benefits of both memory management techniques: we introduce two new operating system functions that can be invoked in the face of a decision to place or decommission an application on the machine. One function implements a swap-in (or "prefetch" in our terminology, to avoid confusion with general swapping) a process into memory. The local controller utilizes this service before issuing the SIGCONT signal to the process for the AppServ being placed on the machine.

The other function swaps-out ("pre-purges") a specified process from memory to disk. The local controller uses this function after suspending the AppServ with the SIGSTOP signal to ensure there is enough memory for active AppServs on the machine and to increase the contiguity of the pages of the swapped-out process on disk.

4. PERFORMANCE

We have implemented the mechanisms discussed above in a simple hosting service. In this section we study application placement performance before turning our attention to agility in the next section.

We use the following setup to study the performance of the application placement mechanisms sketched in the last section. We used a server with a 2.8 GHz Intel Pentium 4 CPU and 512 MB of memory running Linux kernel 2.6.21 as a hosting server. The machine is equipped with two disks, one hosting a large swap area (13 GB) and the other holding the application files. Except where otherwise indicated, applications are run within JBoss Application Servers. While we conduct our experiments mostly with JBoss, we have no reason to believe that other application server implementations might behave differently with our mechanisms.

We use three sub-applications in our experiments: (*a*) the transactional Web e-Commerce benchmark TPC-W[1] with 1000 items intended to represent a realistic application; (*b*) a synthetic application that imposes a tunable CPU load (referred to as CPU-bound application); and (*c*) a synthetic application with a tunable memory footprint (referred to as memory-bound application). The CPU-bound application defines a single request type that contains a loop consuming a number of CPU cycles. By applying a certain request rate to this application from an external HTTP client we can study the performance of application placement schemes in the presence of varying amount of CPU load due to competing applications. The memory-bound application defines two request types. One type initializes a large static array of characters on the first request (the array size is the tunable

[1]http://www.tpc.org/tpcw/default.asp

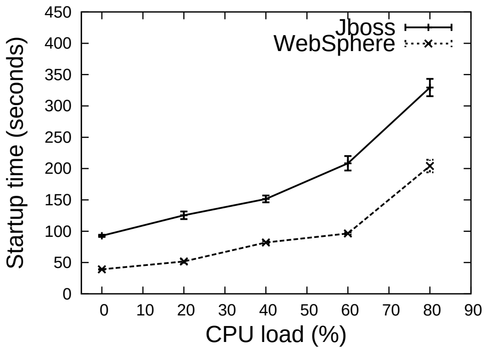


Figure 2: Regular startup time vs. CPU load

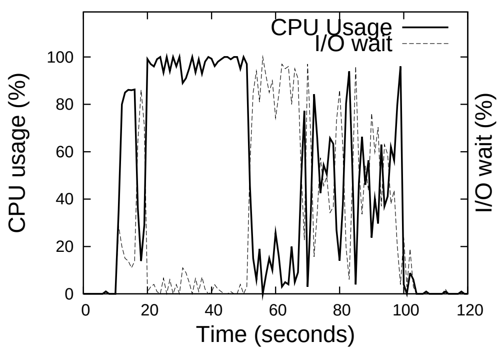


Figure 3: CPU and disk I/O usage dynamics of regular startup.

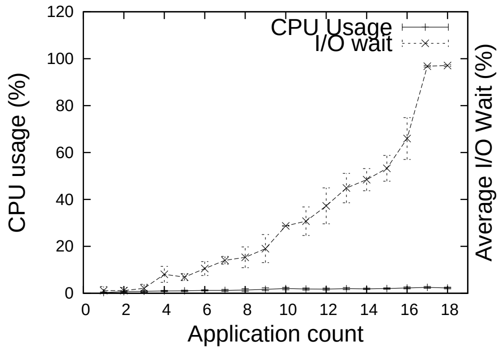


Figure 4: Resource consumption of idle applications.

application parameter), and then touches all those characters on subsequent requests. The other is a null request whose purpose will be explained in Section 4.4.

For convenience, we package all applications into a single JBoss (or WebSphere when it is used) server instance. Unless otherwise mentioned, we use *wget*[2] to generate client requests in all our tests. All experiments in this section are repeated ten times, and the error bars in the graphs show the 95% confidence intervals.

4.1 Regular Startup

We investigate regular startup delay on an idle machine (corresponding to the full-machine allocation alternative as described in Section 2) and in the presence of competing load (representing the shared-machine alternative). We start a copy of JBoss containing all our test applications and apply a certain request rate to the CPU-bound application to generate the desired CPU load. We then start a second instance of JBoss and observe its startup delay. We conservatively measure startup delay as the time between when the local controller receives the *Start()* message and the time JBoss reports that it is ready to process requests. In reality, the startup time can be higher because the processing of the very first request takes longer than the subsequent requests.

Figure 2 shows the the startup delay as a function of the competing CPU load. The startup delay of JBoss is over 90 seconds on the idle machine and increases linearly as the CPU load increases – to 150s at 40% load.To make sure that the long startup time is not an artifact of JBoss implementation, we repeat this experiment with WebSphere AppServ. While the WebSphere startup time is smaller than that of JBoss, it exhibits the same general trend, with almost 50s delay on the idle machine.

Other studies also report long application startup delays that could extend to several minutes depending on the particularities of the application and the server machine [18, 13]. Startup delays on the order of minutes clearly limit the agility of a hosting platform: an overloaded site will have to wait this long even after the platform decides to give the site more resources. In a modern high-volume site, this can translate to a degraded service to a large number of users.

In addition to startup delay, an important aspect of the application placement efficiency is its resource consumption, because it represents an overhead that could otherwise be utilized for processing client requests. We measure this overhead in terms of CPU usage and I/O wait percentage. The latter is an indication of disk usage, and is defined as the percentage of time that the CPU was idle during which the system had an outstanding disk I/O request.

Figure 3 shows the dynamics of CPU utilization and I/O wait time percentage during an AppServ's startup period on an otherwise idle machine. The startup commenced roughly 10 seconds into the experiment and finished 90 seconds later. As shown in the figure, the regular startup process is resource-intensive. For the initial 30–35 seconds of the startup process, the CPU utilization is close to 100%, followed by a period of heavy disk use, and finally a period of intermittent CPU and disk usage. This indicates that starting an AppServ instance can interfere in significant ways with competing applications on the same server.

In conclusion, the regular startup approach is slow and expensive and this hinders the agility possible in a placement algorithm. While we do not delve into particular placement algorithms in this paper we will now turn our attention to techniques that will lessen the burden of application switching, which we believe will enable a new class of placement algorithms.

4.2 Run-Everywhere

We now turn to the run-everywhere alternative. We perform the following experiment to determine the costs involved in this approach. We start a JBoss instance and wait

[2]http://www.gnu.org/software/wget/

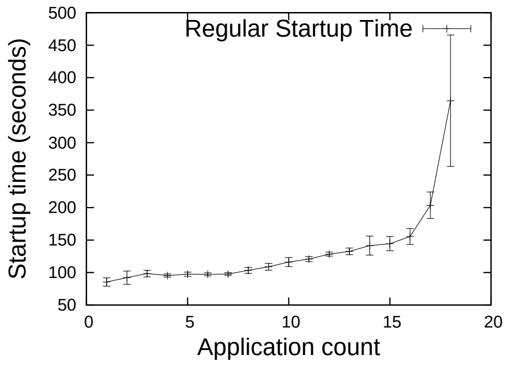


Figure 5: Application startup time vs. the number of pre-existing idle applications.

until it reports startup completion. We then monitor the overall CPU and disk usage on the server for the next 100 seconds before we start another JBoss instance. We repeat this process for as many AppServ instances as possible.

Figure 4 plots the average percentage of CPU usage and I/O wait time as a function of the number of AppServ instances running. The figure stops at 18 AppServs because starting additional AppServ instances beyond that point becomes extremely slow in our setup. Figure 4 suggests that maintaining a large number of idle AppServs on a machine causes disk thrashing. In other words, AppServs do not seem idle to the operating system despite the fact that they are not processing any user requests. The likely reason is that AppServs perform some housekeeping even if they are not processing requests [7]. Although we observe that housekeeping requires little resources in the absence of large number of competing applications, it causes the operating system to spend most of its time swapping processes as the server's resources become increasingly constrained.

By consuming resources, idle AppServs in particular increase the startup delay for new AppServ instances that might not have been included into the initial set. As we start AppServ instances in the experiment described above we tracked the startup time of each of these instances. Figure 5 shows the rapidly growing startup time as the number of AppServs approaches the limit.

Overall, we conclude that starting an instance of every application on every machine and allowing the OS to manage the processes is not feasible since AppServs never become truly idle from the OS's perspective. Consequently, we next explore schemes whereby the OS is explicitly informed about whether the overall platform considers an AppServ instance to be active or idle.

4.3 Regular and Enhanced Suspend/Resume

Suspended processes consume no CPU cycles, and we found that the resident memory footprint (i.e., the memory that could not be forced to page out) of a suspended AppServ is around 700 KB. Therefore, even a modest system can support a large number of suspended (idle) applications at a minimal memory cost (e.g., 100 idle AppServ instances can fit in 70 MB).

To measure the startup time of the regular and enhanced suspend/resume techniques, we perform the following ex-

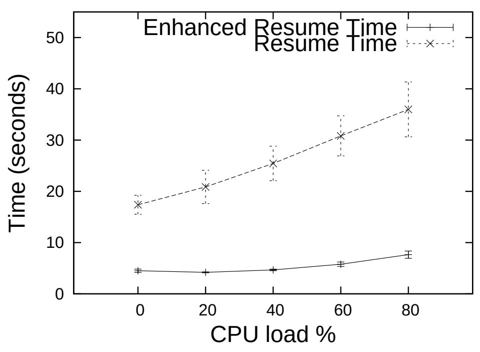


Figure 6: Resume delay vs. CPU load for the suspend/resume approaches.

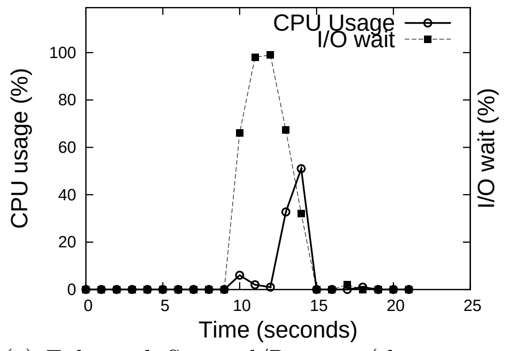
periments. For the regular suspend/resume, we start an AppServ instance from scratch and send it a SIGSTOP signal once the startup is completed. The process is repeated for more AppServ instances with the goal of pushing the earlier instances out of memory. We stop once the resident memory footprint of at least 10 suspended instances decreases to about 4 MB each[3]. Pushing suspended AppServs' pages out mimics an expected usage scenario: with coarse-grained application placement, we expect suspended AppServs to be pushed out of memory. For the enhanced suspend/resume, each suspended AppServ is explicitly pre-purged.

After the above preparation, we resume the suspended AppServs (one at a time) using the suspend/resume and enhanced suspend/resume approaches, and observe their reactivation time. We repeat this experiment with various competing CPU loads by exercising the CPU-bound application in the background.

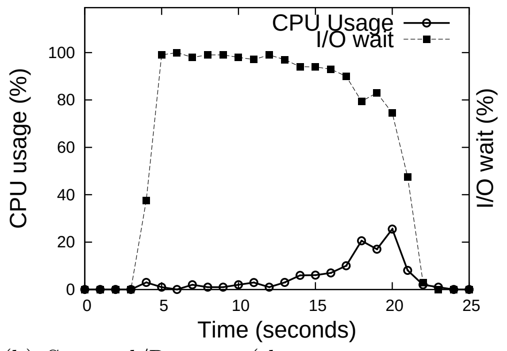
We use the completion of a small application request (the home page of TPC-W) to indicate that the application instance is active. This is a proxy since the AppServs processes do not report they have finished "waking-up" as it was the case for the regular startup (JBoss reports the startup time). Adding this processing time makes our conclusions conservative because they put suspend/resume techniques at disadvantage relative to the regular startup.

Figure 6 shows the results. Compared to regular startup, even the non-enhanced suspend/resume activates AppServs 5.3 times faster on an idle machine, with the advantage growing in the presence of competing load. Furthermore, the figure demonstrates the added benefits of the enhanced suspend/resume approach over the non-enhanced version, especially at higher CPU loads. Similar to the regular startup, the non-enhanced resume delay depends on the CPU load because the non-enhanced resume process requires CPU cycles for page fault processing, CPU scheduling, and freeing memory. However, the enhanced resume time virtually does not change as the CPU load changes since the added enhancements avoid the demand paging overhead. Compared to regular startup (Figure 2), the enhanced suspend/resume is over 20 times faster on the idle machine and over 30 times

[3]Reducing the memory footprint further is possible but requires starting a large number of additional AppServs. Stopping at 4 MB suffices for our study because it puts enhanced suspend/resume at disadvantage by reducing the amount of memory regular suspend/resume must page in on wake-up.



(a) Enhanced Suspend/Resume (the resume starts at around 10s time).



(b) Suspend/Resume (the resume starts at around 4s time).

Figure 7: Resource consumption of application placement with the suspend/resume approaches.

faster at 40% CPU load.

Figure 7 shows the dynamics of resource consumption of an AppServ instance during the resume process. In this experiment, no background CPU load is applied. The CPU consumption of both regular and enhanced suspend/resume process is drastically reduced compared to that of regular startup. As expected, suspend/resume and enhanced suspend/resume operations are disk intensive because they both need to bring a large number of pages into main memory. However, enhanced suspend/resume saturates the disk for much less time; the reason is that, as we will see later, it stores application pages on disk together so that they are brought back faster.

4.4 Prefetching Overhead

The enhanced suspend/resume we studied so far prefetches the entire process state before resuming the AppServ. However, some of the process pages may not be needed for immediate client request processing. In this case, prefetching those pages unnecessarily delays the completion of the application placement. Because the amount of excessive prefetching depends on the particulars of the application and the workload, this section studies the bounds on the potential delay penalty due to this excessive prefetching.

To this end, we used the memory-bound applications which initializes a large static array of characters on the first request. The application then defines two request types: (*i*) one that touches all the allocated characters, representing the extremely favorable prefetching case when virtually *all* prefetched process pages are required and (*ii*) a null request that does nothing, which represents the extreme case of wrong prefetching when *none* of the application-specific memory is needed after the resume (the only usefully prefetched pages are those holding the AppServ itself).

Figure 8 depicts the placement delay for different application sizes under the enhanced and non-enhanced suspend/resume approaches for both extremes. In the beneficial case (Figure 8a), the the non-enhanced placement delay increases rapidly as the application size increases. The delay of the enhanced resume increases much slower, so the benefits of the enhanced resume grow dramatically for large applications. Still, we expected these benefits since all work done by prefetching is useful.

In the extreme unfavorable case (Figure 8b), while the enhanced suspend/resume performs similar to the previous case, the non-enhanced resume delay no longer grows with the application size and in fact drops slightly. This is due to the specifics of our memory-bound application: the allocation of large static arrays pushes out earlier AppServs in large chunks and hence places their memory pages close to each other on disk. Then, the resumed AppServ benefits from normal operating system's read-ahead as it pages in[4].

Even under these extreme conditions, the enhanced suspend/resume retains some performance advantage over regular suspend/resume. Of course, eventually the two curves on the graph would cross; however, an application size of 280 MB (around 130 MB for the AppServ itself and 150 MB the static array size which will be unneeded memory in this case) was not enough for the cross-over to happen. The efficiency of in-bulk prefetching and contiguous placement of pages on disk mitigates the penalty of unneeded prefetching.

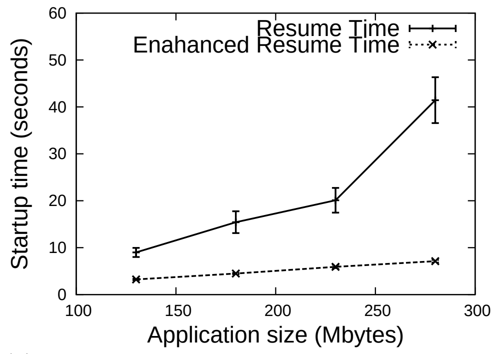
Overall, our conclusion from these two extreme scenarios is favorable to the enhanced suspend/resume: it exhibits dramatic speed-up of application placement over regular suspend/resume when it prefetches useful memory, and retains certain performance advantage even when prefetching large amounts of unneeded memory.

4.5 Contributing Factors

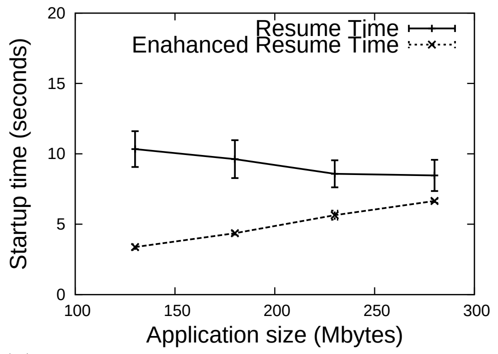
As discussed above, the proposed enhancements decrease AppServ activation in three ways. In this section, we experimentally characterize the contributions of each element to the overall performance enhancement.

To isolate the contribution of prefetching, we disable prepurging and conduct an experiment similar to the experiments that measure application placement delay under regular suspend/resume (see Section 4.3). A number of AppServ instances are started and suspended one by one. Once early AppServ instances are pushed out of main memory, we resume one AppServ by prefetching and measure the resume delay using a small client request as before. Notice that, since we do not prepurge, we do not ensure free memory at the time of prefetching or the contiguity of page placement on disk. Therefore, any performance gain over the regular suspend/resume will be only due to prefetching.

[4]Linux attempts to read a few pages that are adjacent to the requested page to reduce disk head movement.



(a) *All* application footprint is used for subsequent request processing.



(b) Most of the application footprint is *not* used for subsequent request processing.

Figure 8: Application placement delays of regular and enhanced suspend/resume under best- and worst-case scenarios for the enhanced suspend/resume.

Instance #	Number of Disk Areas	
	Prepurging	Page-out
1	31	5139
2	22	5278
3	18	5093
4	18	4897
5	17	4855

Table 1: Contiguity of process pages on disk.

We use the same setup to assess the contribution of ensuring enough free memory at the time of activation. However, before the activation of an AppServ instance, enough memory to accommodate the application instance is freed using a special program that allocates that amount of memory and terminates (leaving the corresponding amount of free memory in its place). No prefetching is performed: the process is activated using a regular SIGCONT signal, and hence it wakes up using normal demand paging. Further, since we do not prepurge the AppServ process, we do not ensure contiguous placement of its paged-out pages on the swap area.

For the third factor — contiguous allocation of memory pages on disk — several AppServ instances are started from scratch, suspended, and then prepurged in bulk one after another. To verify that bulk prepurging results in contiguous disk allocation, Table 1 shows the number of contiguous areas an instance occupies on the swap disk with prepurging and regular paging-out (five different application instances are shown). Prepurged AppServs occupy two orders of magnitude fewer contiguous areas than those paged out using regular paging. After the above phase, several additional AppServ instances are started and suspended without being prepurged, to make sure the memory is filled up. Finally, we resume the prepurged AppServs (one at a time) using a normal SIGCONT signal and measure its resume delay. Any resume speedup will be due to contiguous page location on disk: there is no free memory at the resume time, and the resume occurs using regular demand paging.

Figure 9 shows the result of these experiments, under varying amounts of competing CPU load. For the ease of comparison, the figure also reproduces the curves for regular suspend/resume and the enhanced suspend/resume (which combines all the above factors). The figure shows that each factor results in a sizable reduction of the startup delay, compared to the regular suspend/resume. Prefetching contributes the most to the overall performance improvement, while prepurging and contiguous disk allocation both exhibit similar benefits. Furthermore, combining the three enhancements together results in a cumulative effect, with the most significant performance gain.

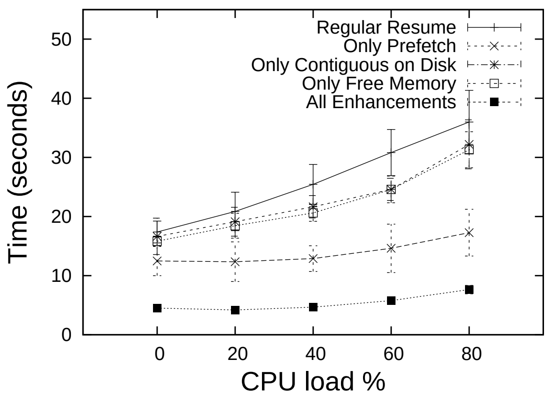


Figure 9: Contribution of different enhancements.

4.6 Effect of Application State

Unlike the current practice of deploying applications from scratch, the suspend/resume techniques restore the before-suspension state of application instances, and the performance of these techniques may depend on the dynamics of the system leading to the application placement (e.g., the application memory footprint, application-level caching, etc). Our previous experiments measured the resume time of AppServs that had been suspended without having processed any user requests. This corresponds to the first placement of a given application on a hosting server. The goal of this section is to consider the performance of our techniques for a repeated placement. To this end, we exercise an application on a hosting server with client requests, then at some

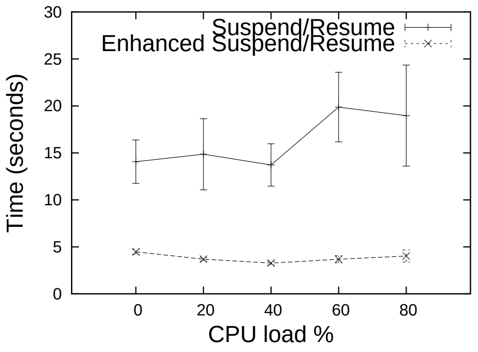


Figure 10: Application placement delay (repeated placement).

point we deallocate it, and then re-deploy it again.

We use TPC-W benchmark, which simulates a bookstore Web site. TPC-W's Remote Browser Emulators (RBE) try to mimic the behavior of real web users shopping at the TPC-W bookstore. We run 30 emulators for 60 seconds against each application instance before it is suspended, and then resume that application in the presence of certain level of competing CPU load.

Figure 10 presents the activation time of both suspend/resume and enhanced suspend/resume. As shown, the regular resume time varies significantly from one trial to another. The likely reason is the random behavior of the RBE's before suspension, with demand-paging bringing varying portions of the application memory footprint back from disk. On the other hand, the enhanced resume time remains stable because it prefetches all the memory footprint anyway. Furthermore, regardless of wide variations in regular resume time, the enhanced resume time remains significantly shorter in all cases.

5. HOSTING PLATFORM AGILITY

Having considered the performance of the various application placement mechanisms on a hosting server, we now study how this performance translates into the end-to-end agility of a realistic hosting platform. We put together a testbed that includes all the components in Figure 1. Our testbed consists of two hosting servers, referred to as the base server which hosts an application initially and the support server where an additional application instance will be activated when the base server becomes overloaded. We use Nortel's Alteon 2208 Application Switch as the request router; the clients send their requests to the switch's IP address and the switch then load-balances these requests among hosting servers at the granularity of TCP sessions.

The measurement module on each hosting server reports server CPU usage to the central controller every second[5]. On detecting an overload (i.e., CPU usage reaching 80% in our experiments), the central controller activates an application instance on the support server by sending it the *Start()* message. The support server deploys the application using one of the mechanisms under study and, upon completion, sends a "ready" message to the central controller. In the regular startup case, the readiness of the application is detected as reported by JBoss. In both suspend/resume cases, it is detected by the local controller sending a special request to the new application instance and obtaining a successful response. Once the central controller receives the "ready" message it configures the Nortel switch to add the support server to the group of servers for load balancing.

We utilize a synthetic application, which comprises a single Java Server Pages (JSP) benchmark page that performs a combination of CPU-, disk-, and memory- intensive operations. The testbed is initialized before the experiment as follows. For regular startup, we pre-deploy (but do not start) the application at the support server. For regular suspend/resume, we start and then suspend the test application, and then start and suspend a number of other instances to make sure the test application is evicted from memory. For the enhanced suspend/resume, we start and then pre-purge the test application.

The testbed is driven by a client workload generator, which generates a series of requests to the benchmark page using *httperf*[6]. Initially, the request rate is low and only the base server is handling the clients' requests. After 50 seconds we increase the request rate (e.g., mimicking a flash crowd). We then observe the dynamics of the average response time as the measure of the agility of the hosting platform.

Figure 11 shows the experiment results. In the first 50 seconds, the base server is underloaded and exhibits response time less than 200 msec in all cases. However, the platform reaction to the overload differs dramatically with different schemes. As shown in Figure 11a, the response time of the platform with regular startup increases to over 40 seconds in some cases. Furthermore, it takes the platform roughly 100 seconds to adapt to the new load.

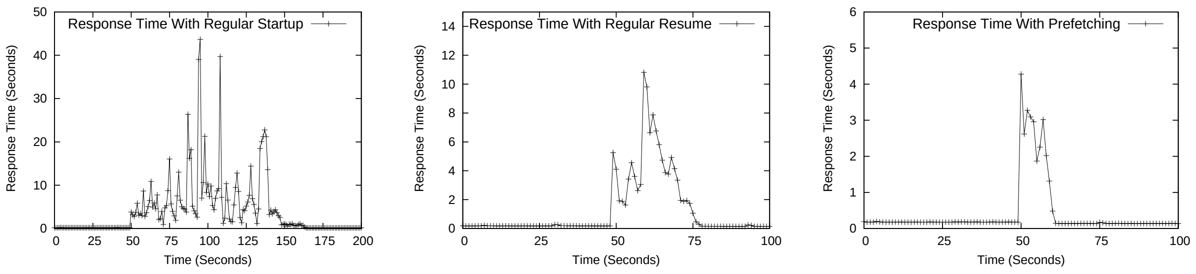
The platform agility increases with regular suspend/resume. Figure 11b shows the period of degraded service decreases to just over 25 seconds, or 25% of the time required by regular startup. The severity of service disruption during the adaption is also reduced, with the response time spiking to only at most 12 seconds, because of a smaller backlog of requests.

Figure 11c shows the response time dynamics of the platform using the enhanced suspend/resume approach. The enhanced suspend/resume further improves the platform agility. It cuts the degraded service time to roughly 11 seconds—or by a factor of 2 compared to regular suspend/resume—and the response time during the adaption to less than 5 seconds.

The bulk of the adaption time in this case is now spent not for application placement but for switch reconfiguration. It took the switch around 7 seconds in all our experiments from the time it is sent the reconfiguration messages to the time requests started showing up at the support server. While this delay can be viewed as negligible in existing application placement mechanisms, it becomes dominant in the enhanced suspend/resume. Further study revealed that much of the switch configuration delay is due to its storing the new configuration on a flash card. We speculate that this design was driven by the fact that load balancing switches were assumed to be used in static environments, where they balance load among a fixed set of servers. It appears that, in dynamic utility computing environments, a better approach would be to start load-balancing before flushing the new configuration in the background.

[5]In reality, measurement modules could also monitor other types of resources, but CPU is sufficient for our experiment.

[6]http://www.hpl.hp.com/research/linux/httperf/



(a) Regular startup (b) Suspend/resume (c) Enhanced suspend/resume

Figure 11: End-to-end agility of a hosting platform with different application placement mechanisms (note different scales on the y-axis).

6. RELATED WORK

There is a growing interest in the efficient management of data centers and hosting platforms. Different approaches target different resource-sharing environments: multiple applications sharing the same AppServ instance [21, 23], multiple AppServ, each dedicated to its own application, sharing the same physical machine [25, 13], each application running in its own dedicated virtual machine, with virtual machines sharing a physical machine (e.g., [10, 2]), and, finally, each application instance running on a dedicated physical machine, so that resource management in the data center occurs at the granularity of the entire machines [3]. These environments differ in their resource sharing flexibility and the degree of application isolation they provide. The immediate target of our work is the environment where application instances are hosted as normal operating system processes on a server machine. Previous work addressing this environment [25, 13] recognized the costs of changing application placement but focused on minimizing the placement changes. Our approach is complimentary to this work in that we are aiming at reducing the delays and costs involved in carrying out placement decisions—which we believe will enable a new class of more savvy decision algorithms.

Placement of application replicas is one of the key aspects in a hosting platform. Previous work in this area has mostly focused on the algorithms for deciding on the number and location of the replicas, either in the wide area network [11, 19, 20, 12] or within a given data center [25, 13, 26]. By reducing the delay in enacting these decisions, our approach should allow for more dynamic algorithms that respond better to sudden demand changes.

The agility of hosting platforms has been recently addressed by Qian et al. [18]. Unlike our work, the authors targeted environments in which application instances are organized in application-level clusters and hosted on virtual machines. To improve the agility of resource reassignment, the authors rely on a limited number of stand-by virtual machines (called ghost VMs) that reside in memory. The benefits of their approach depend on the correct decision on which VMs to run as ghosts. On the contrary, our approach aims to reduce the delay of activating idle AppServ instances that reside on disk. We believe that the ideas behind our approach will be applicable to speeding up the resumption of a suspended VM as well, and plan to investigate this possibility in future work.

We use prefetching to increase the efficiency of the swap-in process. Prefetching has been studied extensively and used in multitude of applications ranging from instruction prefetching in microprocessors to Web prefetching. The literature on prefetching is too vast to cite here (see [16] and the references therein for initial pointers to prefetching in the operating systems context). We particularly used the swap prefetch kernel patch [24] as a base for our implementation.

7. LIMITATIONS AND FUTURE WORK

Our approach can make the effects of bad programming practices more pronounced. For example, application suspension may lead to the timeout of database connections, and proper exception handling is needed upon the resume. Handling such errors is important in any environment, and one might view the fact that our approach brings this need to the fore as a benefit and not a drawback.

A current limitation of our approach is that its benefits would be reduced in an environment where application instances of a given application form an application-level cluster. Several application servers, such as WebSphere and Weblogic, provide this functionality, which allows application instances to share client's session state and fail-over from one application instance to another in the middle of a session. With application clusters, whether an application instance starts from scratch or is resumed from a suspended state, it needs to (re)join the cluster, which currently takes over 30 seconds on a WebSphere [18]. When added to the placement delay of all approaches, the relative advantage of enhanced suspend/resume will become less pronounced. In the future, we plan to explore more efficient application cluster membership mechanisms.

Another direction for future work is exploring more elaborate prefetching policies of process state. The policy we discuss in this paper simply prefetches an entire AppServ process into memory. We have started to investigate a second policy—which we refer to as "snapshot prefetching"—that prefetches only the part of the application that was actually accessed in memory the previous time the application was active. We could consider additional policies such as those derived from an LRU stack.

Looking ahead, it is possible that no single mechanism will prove sufficient for the needs of future hosting platform, and some combination of these mechanisms will be required. For example, although we show that maintaining idle application instances is too expensive using the "run-everywhere" approach, it might be feasible to maintain some number of spare idle applications, which would serve as a "quick-reaction force" to sudden demand spikes and give the platform some time to place additional application instances. By cutting the placement delay time, our enhanced resume techniques would allow the system to cut down on the number of spares required in such an environment. This would be reminiscent of the ghosts approach proposed in the context of virtual machines in [18].

8. CONCLUSIONS

This paper discusses the issue of efficient application placement in a shared Web hosting platform. While existing research in this area has mostly focused on the algorithms that decide on the number and location of application instances, we address the problem of efficient enactment of these decisions once they are made. We show that the prevalent practice of placement enactment by starting and terminating application instances is a time- and resource-consuming process; this bounds the rate at which configuration changes can take place and delays the configuration changes from taking the desired effect. Furthermore, we showed that a straw-man approach to run idle instances everywhere so that they could be engaged at a moment's notice is impractical because these ostensibly idle Web applications actually consume significant resources due to housekeeping activities of their application servers.

Consequently, we explored new mechanisms for application placement. Our best approach involves the cooperation from the operating system and reduces application placement delay by a factor of 20-30 in our experiments: from over 90s to less than 5s on an otherwise idle machine, and from around 150s to still around 5s on a moderately loaded machine. We carefully explored the factors that contribute to these improvements as well as potential limitations of our approach.

Furthermore, we have demonstrated how the above improvements translate into the end-to-end *agility* of the hosting platform [18] in reassigning servers among application instances. By assembling a realistic testbed, we showed that our approach allows the platform to adapt to a sudden demand change ten times faster than when using regular application startup, and with much less severe service degradation during this already-shortened transition period. In fact, we observed that, with our approach, the application placement delay becomes dominated by the time spent reconfiguring the load-balancing switch. We suggested a way to remove much of this reconfiguration delay from the critical path of the application placement procedure.

Besides reducing delays in reallocating shared resources, we believe the additional flexibility afforded by our proposed techniques can open up the space of possible policies for deciding how to allocate resources within hosting platforms.

9. REFERENCES

[1] H. A. Alzoubi, M. Rabinovich, and O. Spatscheck. MyXDNS: a request routing DNS server with decoupled server selection. In *WWW*, 2007.

[2] A. Awadallah and M. Rosenblum. The vmatrix: A network of virtual machine monitors for dynamic content distribution. In *7th WCW*, 2002.

[3] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *12th HPDC*, 2003.

[4] Cisco Distributed Director. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/index.shtml`.

[5] S. Demblon and S. Spitzner. Linux internals (to the power of -1). 2004. http://learnlinux.tsf.org.za/courses/build/internals/.

[6] http://hostcount.com/stats.htm.

[7] JBoss documentation. `https://docs.jbosson.redhat.com/confluence/display/DOC/General+Server+Metrics`.

[8] IBM Network Dispatcher. 2001. `http://www-900.ibm.com/cn/support/library/sw/download/nd30whitepaper.pdf`.

[9] IBM WebSphere Partner Gateway. `http://www-01.ibm.com/software/integration/wspartnergateway/`.

[10] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting utility platforms. In *12th HPDC*, 2003.

[11] J. Kangasharju, J. W. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4), 2002.

[12] M. Karlsson and C. T. Karamanolis. Choosing replica placement heuristics for wide-area systems. In *ICDCS*, pages 350–359, 2004.

[13] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *15th Intl. Conference on World Wide Web*, 2006.

[14] R. O. King. SMB hosting market still undeveloped. Web Hosting Industry Review. `thewhir.com/features/smb-undeveloped.cfm`, 2004.

[15] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *ICDE*, 2004.

[16] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. In *HotOS*, 2005.

[17] Performance characteristics of virtualized systems with the VMware ESX server and sizing methodology for consolidating servers. IBM White Paper. `ibm.com/websphere/developer/zones/hipods`, 2007.

[18] H. Qian, E. Miller, W. Zhang, M. Rabinovich, and C. E. Wills. Agility in virtualized utility computing. In *2nd VTDC*, 2007.

[19] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001.

[20] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *ICDCS*, May 1999.

[21] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating Internet applications˙In *Proc. of the 8th WCW*, Sept. 2003.

[22] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: autonomic data replication for web applications. In *WWW*, pages 33–42, 2005.

[23] S. Sivasubramanian, G. Pierre, and M. van Steen. Replicating web applications on-demand. In *SCC*, Sept. 2004.

[24] Swap prefetching. https://lwn.net/Articles/153353/.

[25] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW*, pages 331–340, 2007.

[26] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, 2002.

[27] Yankee group's SMB survey suite offers the most comprehensive view of the SMB market. CRM Today. `http://www.crm2day.com/content/t6_librarynews_1.php?news_id=116800`, 2005.