# Xen-ophobia: On profiling boot startup

Theophilus A. Benson, Steven Kappes
*tbenson, kappes@cs.wisc.edu*
*Computer Sciences Department*
*University of Wisconsin, Madison*

May 15, 2008

## Abstract

*The fact that systems fail should come as no surprise to anymore who has ever developed or worked on a system.* A failure reduces the availability of the system and hence the productivity of entities using this system. To increase system availability, several approaches have been developed; these approaches range from simple techniques such as restarting the entire system, to complex algorithms that isolate and restart the failed subsystem. We find the simple approach of a system reboot to be particularly interesting because it is a widely deployed approach. We profile the set of instructions excuted during the startup phase of the system and identify heavily utilized segments. We implement, in Xen, a framework for both monitoring the start-up sequence and identifying highly utilized segments of code. We show that modifications to the identified segments affect the start-up sequence. Finally, we examine the identified segments and suggest modifications that will, if implemented, increase availability by reducing the boot up time.

## 1 Introduction

The system restart plays a key role in the recovery model for many large and popular systems. For example, The Blue Screen of Death or system failure, is perceived as the default recovery behavior for failures in the Windows O.S. [4]. In the famous Blue Screen of Death scenario, Windows panics and prompts the user to restart the system. System reboot as a recovery model is not native to Windows Operating Systems, in [7], Prabhakaran et al. show that the system reboot model is employed by several Linux file systems. There are many reasons for the prevalent usage of this model. The reasons range from simplicity of implementation to negligible overhead. Although simple, this model is not without its drawbacks; it forces the system to perform the entire boot-up sequence. The boot-up sequence is often time consuming as it loads and initializes modules and drivers required by programs that use the system. The amount of time spent executing the boot-up sequence directly impacts the availability of the system; system availability is the

$$\frac{MeanTimeToFailure}{MeanTimeToFailure + MeanTimeToRecovery} \quad (1)$$

Mean time to recovery is essentially the amount of time spend executing the boot up sequence and it follow from eq(1), that optimizating the boot up sequence will increase the system availability.

In this paper, we pose the following question: *Is it possible to identify a small portion of the boot-up sequence which if optimized will result in significant improvements in the availability of the system?* To answer this question, we design and implement a framework to profile the boot-up sequence. This framework utilizes pc sampling to identify the distribution of time spent in each segment of code executed during boot-up. The framework

identifies and ranks code segments based on the frequency with which we sample them.

With the framework in place, we develop and apply a ballooning technique, to increase the amount of instruction executed in a segment. The goal of ballooning is to quantify/validate the achievable gains from the optimization of the identified regions. We show that by ballooning and increasing the number of executed instructions we increase the boot-up time and the rank of the modified segment. We claim that the inverse of this holds; deflating a function or decreasing the number of executed instructions should reduce boot-up time and the rank of the modified segments. Upon examination and analysis of the top 5 segments identified by the framework, we developed a few suggestions for reducing the cycles spent executing boot-up sequence. Finally, we validate our the framework by analyzing the profiles of certain code segments over different boot-up sequences.

The rest of this paper is as follows: section 2 presents a brief literature survey of the domain space. In section 3, we discuss the implementation of our framework. We then present and evaluate the results of running our framework on a real linux operating system in 4. Finally we conclude with a summary of our achievements in section 5.

# 2 Related Works

Our work builds on [7] which identifies failure models employed in various portions of the file system code. Our work looks into increasing the availability of systems that employ system restart recovery model. We believe that system restart recovery is the most widely applied failure recovery model in the systems community. The systems community has worked for many years on profiling techniques to identify code which requires optimization. Most of the work in the profiling space focuses on user-level applications [2], and kernel code [1]. Recently, however, [6] presented a seminal approach to profiling virtual machines.

Our current approach differs from [2], in that our framework doesnt discriminate between user level processes and treats all user level process as one. Unlike our architecture, the architecture presented in [2] targets specific processes and only analyses time spent in user space.

Unlike, prior kernel profilers [1] which require the kernel to load certain modules before profiling can be initiated our approach can begin profile from startup time. In comparison, our method suffers a major draw back because we only profile kernels that have been modified to run on the xen virtual machine. These modifications alter the shape of our results and add noise that would otherwise not be present in the environment profiled by other kernel profilers. Contrary to the approach taken in [1], we perform software level monitoring.

Finally, seminal work in [6] presents a framework similar to ours, however, we defer in two things; where profiling analysis is performed and the amount of data-structures used. [6] runs profiling anlysis tools from within the profiled operating system while we run our tools from the main operating system running in domain 0.

# 3 Architecture

Our profiler implementation requires changes in three different levels: the Xen Hypervisor, the Linux kernel, and user-level tools. This implementation supports flexibility and ease of use with regard to acquiring profiling data.

## 3.1 Program Counter Sampling

A useful way to determine a performance profile is with program counter sampling. The program counter contains the address of the next instruction to be executed. By reading this value, it is possible to determine what code is currently running.

Therefore, this value can be sampled many times to determine where in the code most of the time is being spent. For example, if two

functions are always called in sequence yet one function shows up in the profile more often, it can be inferred that this function is consuming more processor time than the other.

Consequently, program counter sampling is an unobtrusive way to determine where the performance bottlenecks of a system are. Source code is not required and does not need to be analyzed to determine this. However, a symbol table compiled with executables is necessary to correlate a program counter value to functions in an executable.

### 3.2 Xen

Since the motivation behind our project is to discover performance information of the operating system startup, we required an implementation that could profile performance over the entire startup process. As a result, building our profiler into a virtual machine monitor was a natural choice; the virtual machine gives us the freedom to sample P.C. while various key subsystems are being brough online. We choose Xen as the virtual machine monitor to build our profiler on.

Xen is an open-source virtual machine monitor that runs directly on hardware as the kernel [3]. Each guest operating system running on top of Xen is referred to as a domain. Domain-0 is the first guest operating system, which receives special privileges. These privileges include scheduling other domains, multiplexing hardware devices, and providing a user-interface. This allows the Xen Hypervisor to be as simple as possible.

Most of our modifications to the Xen Hypervisor are in the timer_interrupt function. In this interrupt handler, we profile the guest operating systems. The high level pseudocode for this operation is presented in Figure 1:

```
domainID = getCurrentDomainID();
if (domainID > 0) {
pcBuffer.record(currentPC);
}
```

Figure 1: Xen Hypervisor Profiler: pseudocode for profiling Program Counters in Xen

The program counter buffer is implemented as a circular buffer, where the buffer overwrites the oldest values if it becomes full. We did increase the frequency of the timer_interrupt in the Xen Hypervisor from 100 ticks a second to 1000. This allowed us to get more program counter values in a single run and reduce the total number of runs required to get a large sample. Program counter values originating from Domain-0 are not sampled. Only those P.C.s generated by unprivileged domains are sampled. The reasoning for this is explained in the usage section.

Xen improves virtual machine performance by allowing the guest operating system to know they are being virtualized. Therefore, the guest operating system can make upcalls to the Hypervisor known as Hypercalls. A Hypercall is very similar to a system call, other than it uses interrupt 82 instead of interrupt 80. By directly calling into the Hypervisor, the number of context switches between the guest operating system and Xen can be reduced.

Consequently, we added a Hypercall to Xen to allow the guest operating system to retrieve the program counter values from the buffer. The Hypercall simply copies the values in the program counter buffer to the buffer supplied by the guest operating system. The pseudo-code for this operation is presented in Figure 2:

```
PCValuesLoaded = 0;
Foreach entry in the supplied buffer {
If (program counter values exist) {
CopyPCValueToBuffer(pcBuffer.Get());
PCValuesLoaded++;
}
}
EmptyBuffer(PCValuesLoaded);
Return PCValuesLoaded;
```

Figure 2: Xen Hypervisor Profiler: Pseudocode for retrieving Program Counters

The Hypercall returns the number of program counter values retrieved. All the program values that are copied to the buffer supplied by calling program are then cleared from the Hy-

pervisor's buffer.

### 3.3 Linux

In order to support profiling, we made several Changes to the Linux kernel. In our framework, analysis of the samples is performed in user realm while the samples themselves are stored in xen realm. To facilitate analysis of data we added a system call to Linux. This system call simply forwards the request to the Hypervisor via a Hypercall. The Hypercall fills the provided buffer with program counter values.

### 3.4 User-level

The final piece of our framework is the user-level application which analyzes the data received. The first user-level program acquires the entire program counter values currently stored in the Hypervisor. This process repeatedly calls the system call added in 3.3 until all the program counter values have been retrieved. When the user-level program calls the system call, the interrupt is trapped by the Hypervisor, which then transfers control to the Linux kernel. The Linux system call then transfers control back to the Hypervisor with the Hypercall. The same process happens in reverse when transferring control back to the user-level program.

Once the program counter values have been retrieved, a variety of scripts are run to interpret these results. The first task is to map program values into the three different realms, the Xen Hypervisor, the Linux kernel, and the user-level. Memory Addresses is partitioned into three different realms. The addresses belonging to the Xen Hypervisor are all greater than 0xfc (in hexadecimal). Values that originated from the Linux kernel will be greater than 0xc (in hexadecimal). Any address value below the kernel boundary belongs to the user-space.

In addition to mapping values to the different realms, we also map program counter values to functions in both the Xen and Linux realm. The symbol tables for both of these realms are provided, so we just need to check the specific range the program counter value falls in to determine the function. The user-level program counter values cannot be mapped to the function level, because this requires knowledge of the exact process from which the P.C. was sampled.

Once program counter values are mapped to realms and functions, the most important information is to get the profile from the data. The profile contains the frequency for both the realms and the function over the sample data. We consider both the number of times sampled and the percentage of total samples. We get the profile for both aggregate frequencies in realms and frequencies in individual functions.

The profile of the sample is important because this contains the information on where optimization should be done. For instance, if one function is accounting for over 50

In addition to determining the frequency profile, we also get frequency information based on time. This allows us to see how much a function is being called for a specific time interval. One use of a time frequency profile is to validate that multiple executions produce similar results in terms of when functions are heavily called.

### 3.5 Limitations

One limitation of this architecture is that user-level applications cannot be profiled. All program counter values in the user-space are aggregated into a single entry in profile results. This is due to the fact that user-level programs will have overlapping program counter values. Since our profiler is built into the Xen Hypervisor, it is not possible to determine the process that is running. We could have queried the Linux kernel to derive this information, which would allow us to determine the frequency information for individual processes. However, none of the user-level executables we checked included symbol table information. This combined with the fact that we were profiling startup, which does not spend a lot of time in user-space, we decided this was of little utility. By modifying Linux of the unprivileged domain, it also may have affected the profile.

The fact that Xen uses paravirtualization also affects the profile returned by our architecture. Since the guest operating system is modified to make Hypercalls, this will return a different profile than a Linux install running directly on the hardware. This limitation is visible in our results.

Finally, operating system startup in a virtual machine is not identical to startup directly on a physical machine. For instance, Xen must emulate BIOS since the guest operating system cannot access these basic devices directly. We are unsure of how this affects the profile results since we cannot get profile information without Xen.

### 3.6 Usage

With this framework in place, we can record program counter values from any non-privileged domain. Then, we can run our user-level tools in the privileged domain. Then, perform whatever operation to record profile information in the unprivileged domain. The program counter values then can then be accessed in Domain-0.

This makes it easy to profile startup. First, the program counter values in the Xen Hypervisor should be cleared. The unprivileged domain can be created from Domain-0. Once the unprivileged domain is finished starting up, it can be killed. The program values retained in the Xen Hypervisor will correspond to the entire startup process.

## 4 Analysis

In this section, we present the environment in which our experiments were performed and methodology used to perform them. Then we discuss the repercussions of our results and conclude by summarizing insights gained.

### 4.1 Experiment Setup

We profile the boot-up sequence for the Ubuntu 6.06 Linux Disto [5]. The variant of ubuntu that we profiled was modified to allow for integration with the xen hypervisor. We built the hypervisor from the xen 3.0 code which we modified to instrument our architecture. Our tests were performed on two sets of computers; these computers had slightly different hardware. We now present that parameters for the machine on which most of our tests were performed; this computer was equipped with a Pentium 2.33 Core 2 duo processor, 2 GB DRAM , and 160 GB HDD(7200rpm, 12ms seek).

### 4.2 Experiment Procedure

In our experiments, we define the boot-up sequence as the set of instructions executed from the time power comes on until the user receives a login prompt. We limit our profile to the boot-up sequence by manually monitoring the guest operating system and stopping the user level profiler once the prompt comes up.

Using the profilers dump as input for our perl script we resolve the P.C. values to Symbols. The frequency and distribution of these symbols are graphed and presented in the next section. We note that this manual step introduces some error; we sample a few P.C. values from after the boot-up sequence. We ignore the effects of the extra P.C. values because they are statistically insignificant: 10 extra values do not affect the distribution in a sample of roughly 25,000 values.

### 4.3 Experiment Results

In this section, we present two groups of results, the first displays the percent of time spent in the realms. The second presents the portion of time spent in the sampled symbols of each realms.

In Figure 3, we present the percent of time spend in each of the three realms over the course of the boot-up sequence. From this figure, we observe that the overheard of virtualization is significant and relatively constant; time spent in the xen realm accounts for virtualization overhead. Unlike the profile for the xen realm, we notice that user and kernel realm both present interesting and distinct profiles. In Fig-

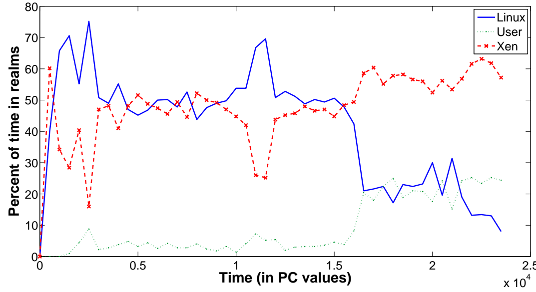


Figure 3: **Distribution of time spend in User,Kernel, Xen Realms.** .

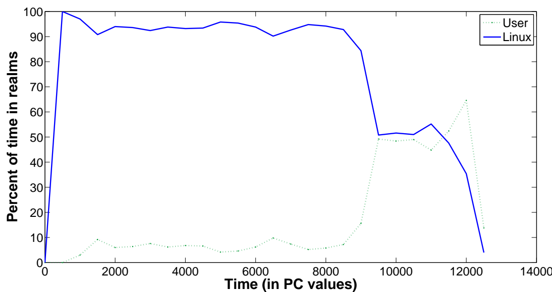


Figure 4: **Distribution of time spend in User,Kernel Realms.** .

ure 4, we graph the profiles for the two realms, user and kernel, without the noise of virtualization. In this new graph we notice that time spent in user space is relatively insignificant, averaging 3%, until the last mile of the boot-sequence. During the last mile the time spent in user realm more than doubles that spent in kernel realm.

We believe that the kernel and user profiles can be explained by the structure of the Linux bootup sequence. In linux, the first process created after suitable initialization of hardware is the init process. This process runs in user realm and parses several configuration files during the boot-up sequence. The init process uses the contents of the configuration files to load and configure modules. The loading and configuration takes place in kernel space. For example, the configuration file for networking contains the interface to bring up and the parameters for this interface; while the configuration file may be read in user realm jumps need to be made to kernel realm to access privileged instructions that deal with the ethernet driver.

Switching between user and kernel realms requires the use of a system call. In figure 6, we present the profile for the system_call symbol over the course of the boot up period. We find that system_calls are sampled frequently and consistently through out the boot-up sequence. These findings lead us to propose the following question: *Are symbols correlated? Is there a sequence of symbols that are sampled repeatedly?* In response, we developed perl scripts to correlate symbols and we find one such sequence of correlated symbols; **scrit system_call scrit system_call**. An interesting observation from this sequence is that at several points during boot-up, code is essentially switching back and forth between realms. The sequence is generated from 1 millisecond samples and while a lot could indeed occur between these samples, we still find it interesting that the sequence occurs frequently.

In our correlation analysis we notice that scrit and system call symbols are sampled frequently, in Table 1 we present the top 10 most sampled symbols. Of the 10 symbols presented, we noticed that only the top 6 are relevant as these account for over 70% of the cycles used during boot-up. Of these 6 symbols, 5 are used for context switching either between the user and the kernel or between the kernel and xen realm. We notice that the more frequent switch between kernel and xen account for 40% of the cyles lost during boot-up sequence. In contrast, the user user to kernel switch accounts for half, or 14%, of the time spent during boot up.

From these results we conclude that the cost of virtualization is significant and that optimization of the context switching code will provide faster boot-up sequence.

### 4.4 Experiment Validation

In the previous section, we observed that an operating system running in a virtual machine wastes approximately 60% of its time switching between the various realms. Also we note that an operating system on physical hardware would loose 30% of its bootup time to context switching. From these observations, we con-

| Realm | Symbol Name | Fraction of Time Sampled | Symbol Rank |
|---|---|---|---|
| **Xen** | handle-exception | 0.28891 | 1 |
| **Xen** | hypercall | 0.16096 | 2 |
| **Xen** | hypercall-page | 0.12231 | 3 |
| **User** | | 0.09183 | 4 |
| **Kernel** | scrit | 0.07392 | 5 |
| **Kernel** | system-call | 0.06101 | 6 |
| **Kernel** | page-fault | 0.04858 | 7 |
| **Kernel** | error-code | 0.03929 | 8 |
| **Xen** | flush-area-local | 0.02421 | 0 |
| **Kernel** | paging-init | 0.00968 | 10 |

Table 1: The Top 10 Symbols sampled. For each symbol the realm, fraction of time sampled, and its ranks are displayed in the table

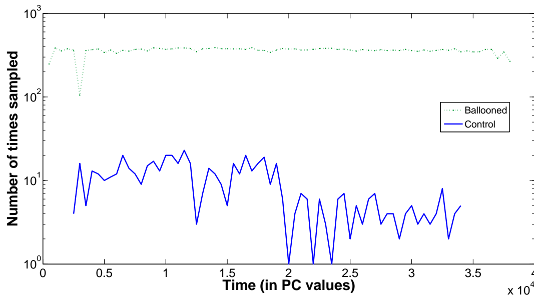


Figure 5: **Frequency Profile for flush area local** *Compare the frequency of the balloned flush area local in blue against the frequency of the normal, unballooned flush area local.*

clude that modifying the switching code would greatly improve bootup time.

Modifying kernel switching code requires some level of familiarity which we currently lack. In the face of inexperience, we show that modifying the number of executed instructions in a code segment changes the boot-up times and affects the frequency with which the associated symbol is sampled. Instead of optimizing a symbol, we slow down or balloon the symbol. Our Ballooning technique consisted of adding a for-loop which performed trivial math calculations. In figure 5, we show the results of ballooning the flush_area_local symbol. The flush_area_local symbol was chosen for ballooning bcause it is one of the less frequently sampled symbols. Figure 5, shows that the ballooned flush-area-local is consistently sampled 3 orders of magnitude more than the normal flush_area_local. In Table 2, we show that boot-

| Version of Flush Area Local | Fraction of Cyles Sampled | boot-up time | Fraction of Times Sampled |
|---|---|---|---|
| **Control** | 0.019 | 30.1 | 0.6 |
| **Ballooned** | 0.719 | 94.2 | 67.5 |

Table 2: Percent of time flush area local is sampled and the amount of time (in secs) taken to complete the bootup sequence

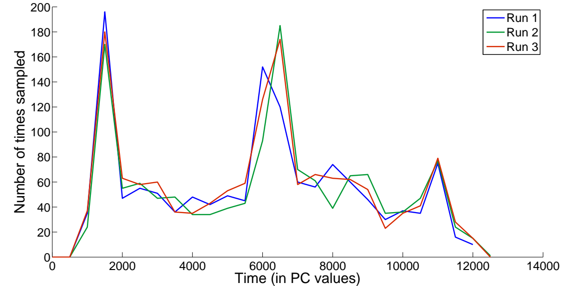


Figure 6: **Frequency profile for System call.** *Frequency of profiling the system call symbol over three different runs.*

up lasts 30.1 seconds for normal runs but lasted 94.2 seconds for the ballooned runs. From The numbers in Table 2, we see that in normal boot up flush_area_local takes 0.6 seconds and onced ballooned it takes 68 seconds. The amount of time not spent in flush area local varies by 2 seconds, with control spending 29 seconds everywhere else and the ballooned run spendign 27 seconds everywhere else. We believe this difference of 2 seconds isn't large enough to invalidate our theory; from these results we conclude that reducing the number of executed instructions will have the desired effect of optmizing the bootup sequence.

### 4.5 Experiment Verification

In this section, we provide verification of our framework in two ways; first, by showing that our results are reproducible. Second, by showing that the symbol profiles recorded align with expected symbol usage patterns. In figure 7, we show the profile for the page_init symbol across 3 randomly chosen runs. We see that in all three runs the function has the identical profile. In figure 6, we display the profile for the system call symbol from the same 3 runs as those used

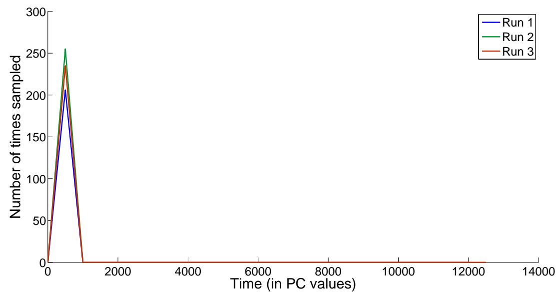


Figure 7: **Frequency profile for page init.** *Frequency of profiling the page init symbol over three different runs.*

in Figure 7. We notice some minor variations in the profiles but nothing significant enough to raise questions in the reproducibility of our approach. In figure 7, we see that the profile for the page_init function aligns well with how we expect it to be used. Page_init, initializes page related datastructures and should ideally be run before paging is used. We find that in figure 7, the symbols are used during the initial 1000 cycles and then never again.

## 5 Conclusion

Fast recovery from failures is crucial in profit or productivity yielding systems. Different systems employ different and sometimes appropriate failure recovery models. One of the prevalent failure recovery models is the system restart model. In the system restart model, the availability of the system is inversely proportional to the time required to execute the boot-up code. We make two contributions to the domain of failure recovery; (1) we implement a framework that identifies critical code segments in the boot-up sequence; (2) we use the framework to identify the critical segments in a real Linux distro and suggest a few ways to speed up the boot sequence.

## 6 Acknowledgements

The authors would like to thank Haryadi Gunawi and Remzi Arpaci-Dusseau for their invaluable guidance and feedback. The authors would also like to thank the Xen Developers, the Ubuntu Developers, and the Linux Developers for their prodigious documentation.

## References

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone, 1997.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] http://www.microsoft.com/technet/. Windows nt: Demystifying the 'blue screen of death.

[5] http://www.ubuntu.com/. Ubuntu drapper.

[6] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM.

[7] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, 2005.