MicroswiftOS : A Minimal OS for Cloud Computing

Menghui Wang
Department of Computer Sciences
menghui@cs.wisc.edu

Yiran Wang
Department of Computer Sciences
yiran@cs.wisc.edu

Chaowen Yu
Department of Computer Sciences
ycw@cs.wisc.edu

Junhan Zhu
Department of Electrical and Computer Engineering
jzhu84@wisc.edu

ABSTRACT

1. INTRODUCTION

Cloud computing launches a new era of computer systems - new service models, such as Infrasturcture as a Service (IaaS), providing computing resources for customers to deploy their own applications. This brings new challenges to both ends of the cloud: for software applications, it is worth deliberating how to migrate multi-tier applications to provide scalable and reliable services[1]; for operating systems hosting client applications, there is much room for optimizations to achieve smaller system size as well as better performance. Since IaaS providers often offer virtual machines rather than physical ones, quite a few features that a regular operation system provides become unnecessary or unimportant, which leaves us opportunities for optmizations.

In modern data centers, a cloud infrasturcture usually consists of many physical host machines, each hosts a various number of guest virtual machines where user applications reside. Such infrastructure is widely adopted, because it helps cloud service providers make best use of their hardware resources, comparing to the alternative where a set of dedicated hardware is provided to each user. The use of virtual machines opens the door for guest OS optimization: (1) Hardware related features are hardly useful in the guest operating system, for it is only running on virtual machines - for instance, support for hot-pluggable device is never useful on a cloud virtual machine. (2) Cloud service providers would like to cut down the size of guest operating systems for storage efficiency. (3) Smaller-sized operating systems are easier to migrate to another physical machine, taking less time and bandwidth. (4) With refined analysis and careful investigation into the kernel modules, removing or modifying certain parts of the OS would not do harm to its performance.

CS 736, Feb-May, 2015, Madison, WI, USA.

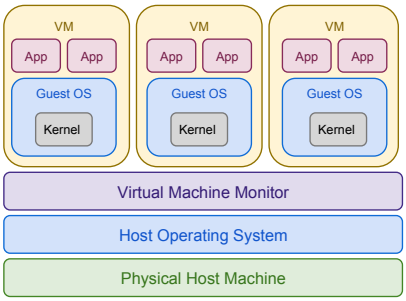

Figure 1: Microswift OS structure

In this project, we try to minimize the Linux operating system to accommodate the requirements of a user machine in cloud. In particular, we aim at guest operating systems running in virtual machines, since most cloud service providers today would give their customers virtual machines that they have complete controls over.

Moreover, the guest operating system could be optimized to support the hosted applications in achieving better availability and performance. [4] For most applications deployed on the cloud, one of the major issues that operations staff care about is the average downtime. In practise, application downtime is inevitable due to (1) preventive system maintainance, (2) application install or upgrade deployment, (3) application failures or outages. In each case, a large amount of time is spent on operating system booting up or rebooting. Therefore, it is important to shorten OS boot-up time to help improve application availability. There are some methods already discovered to speed up Linux booting - disabling unnecessary kernel modules, using a text-based login instead of a graphical login, etc. With a rapid-boot system, our cloud users are able to bring their applications back alive shortly after they experienced a program failure or finished new deployment.

In this project, we make an effort to speed things up during system boot time besides minimizing the size of a regular Linux system. Our goal is to tailor the Linux operating sys-

tem to the needs of a guest OS running on virtual machines in the cloud, and to optimize the micro-OS for swift boot-up. We test our minimal OS for essential features that support common user applications in cloud, such as a simple web server, a scientific computing task, and basic communication services etc. [2]

The rest of this paper is structured as follows. Section 2 captures the characteristics of a kernel running in a cloud, explaining why kernel minimization is feasible and plausible. Section 3 analyzes how we managed to strip down the kernel and obtain MicroswiftOS. It covers several aspects from kernel configurations to boot script optimizations. The implementation details are in Section 4. And Section 5 summerizes the evlauation of MicroswiftOS including system speed, system size and system completeness. Section 6 introduces several interesting papers related to our work. Finally Section 7 concludes our paper. We also include appendix section to elaborate on how we obtain the testing data, and justify the fairness and credibility of our evaluation.

2. MOTIVATIONS

Nowadays, an increasing number of companies and individuals choose to use infrastructure as a service (IaaS) platforms to host their applications, such as Amazon AWS, Microsoft Azure, and Google Cloud Platform to accomplish various tasks including web serving, scientific computations, data storage, application request handling, etc. IaaS customers usually have access to a virtual machine hosted on a physical machine owned by the service provider. Customers can choose their favorite operating system from a preset list, which usually includes some popular Linux distributions and Windows, and they will be able to get full privilege in the operating system they have chosen. Compared to using their own hardwares, IaaS provides customers with a more reliable, flexible, and cost-efficient solution.

Workloads carried out on the cloud have their special characteristics that are different from ordinary workloads expected on a personal computer.One of major characteristics is that a virtual machine on the cloud is usually dedicated to one certain task, such as web serving. Various tasks are deployed on different virtual machines, providing seperation between applications. When one application encounters a fault, we can easily recover the error by restarting the application or the virtual machine without interfering other running applications.

Secondly, the availability of the applications matters. Customers who pay for public cloud service usually come to this decision as they care about the jobs running on the cloud, and would like to seek better support for high availability. Some user applications, such as the web server of a popular online shopping site, may cause huge profit loss if they are unavailable for a period of time.

Thirdly, applications on the cloud do not care about hardware-related features provided by the OS. Cloud service customers do not have physical access to the machine hosting their applications. Therefore certain operations would never take place in IaaS scenarios, such as installing new hardware, or inserting a USB flash key. In addition, as applications will always run on top of a virtual machine, the OS chosen by the customer does not have real control over the hardware of the host machine, but only sees virtualized hardware presented by the Virtual machine monitor.

All of these bring up new opportunies for us to optimize the guest operating systems running in virtual machines to better serve the purposes of the workloads. We want to build a specialized operating system to use in cloud computing with shorter boot-up time and smaller size, and it still meets the requirements of applications runninng on top of it.

2.1 Swift Boot-up

The availability of an application is calculated by

$$\frac{MTTF}{MTTF + MTTR}$$

where $MTTF$ is mean time to failure for the service and $MTTR$ is mean time to repair the failed service. When a failure occurs, usually the time spent on rebooting the system contributes a big portion of $MTTR$. This is especially true when certain monitoring services are deployed, in which they can monitor and detect failures, and in case a failure happens, it promtply reboots the failed machine. Therefore reducing the boot-up time of the OS can help increase the availability of applications running in the cloud.

There are a lot of opportunities for us to optimize the boot-up time. For instance, users and applications on the cloud are less interested in hardware-related functionalities, such as device management and disk utilities. Many modules and can be removed from the kernel, e.g., RAMDisk.

2.2 Minimized OS Size

There are two major benefits of minimizing the size of the guest OS. Firstly, since a physical machine in the cloud will usually hosts many virtual machines, we will be able to save a lot of space if we are able to cut down the size of each guest OS a little. Second, a minimized OS size also enables more efficient migration because of the reduced transmission cost from one physical machine to another.

Since most virtual machine will only run one specific task, rarely an application will need a wide collection of software, as provided by most Linux distributions. Even though, MicroswiftOS still provides a complete toolchain to users to build and install software, so that when a software in need is not included by default, the user will be able to add it to the system manually.

3. MICROSWIFTOS: ANALYSIS AND METHODOLOGIES

MicroswiftOS is an operating system based on Linux and it is optimized for use in cloud computing. It is designed to run on virtual machines. It features fast boot-up and a relatively small image size.

3.1 Optimizing Kernel Configurations

The OS kernel of MicroswiftOS is Linux 3.19, though various changes have been made to it in an effort to speed up booting and shrink kernel size. The kernel features a full-fledged configuration and many modules that could be conviently added on, but most of these features are not used in cloud computing at all. For exapmle, the guest operating system mainly focuses on computing and does not resort to all kinds of peripherals. Some related features such as all-inclusive file system support could be disabled. Some debugging features such as early printk could also be disabled as once our guest OS starts service we assume it does not suffer substantial debugging. Kernel modules could be completely removed because in a cloud computing environment most kinds of services are predefined and ad-hoc functionality changes should not occur. These features and functionalities can be configured before compiling the kernel thus we can prevent the corresponding code from being compiled in the first place. Our first attempt is to find a minimal set of features to be involved in the compiling phase that guarantees sucessful system start up and shut down. By manupulating these configurations, we find a list of features that could be disabled in addition to those already being disabled in default configuration, see Appendix [].

To understand the behavior of operating system bootup, we use perf - a commonly used statistical profiler in Linux - to collect sample data on the execution of a User-mode Linux (UML) and to conduct dynamic program analysis[3]. Since we mainly care about the activities happening at system booting, we use perf to profile a process that does the following tasks:

- Boot UML;
- Log in Linux;
- Halt

We use FlameGraph to visualize the profilling reports - the x-axis spans the sample population, sorted alphabetically, and the y-axis shows stack depth. The function beneath a function is its parent, just like a stack traceback. The first graph is generated from the sample data collected by perf while we execute User-mode Linux with default configurations; and the second graph is generated from that of a minimized UML kernel. When we try to analyze the profiling data we have collected, we encountered a problem of missing symbols and we have not yet figured it out till now. Therefore some of the samples are only reported with their memory addresses instead of commands in the graphs above. But we can still gain some insights from their overall behaviors: for instance, with default configurations the Linux system spends a large amount of time working on something unsymbolized, as shown at the left most part of the graph; while the minimized UML kernel does not have a corresponding timespan.

3.2 Optimizing System Configurations and Software Environments

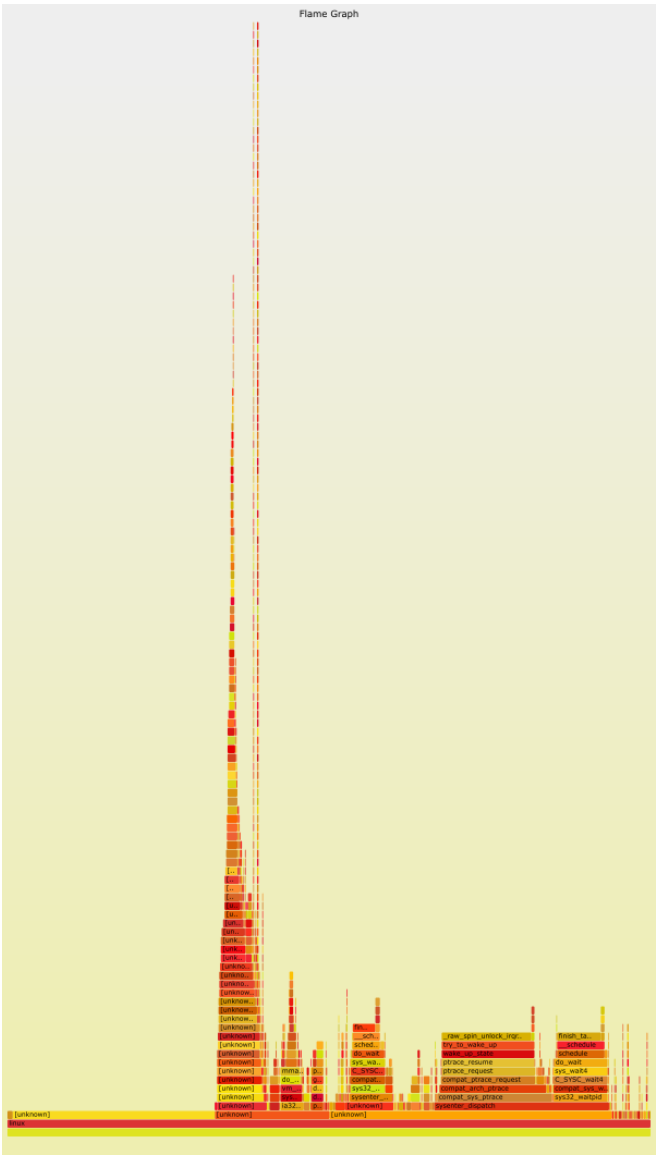

Figure 2: Flame chart for default configuration

Apart from the kernel, an operating system will also need a software environment to run. Pre-installed softwares and configurations will serve for a number of important purposes:

1. Providing a file system on a partitioned disk that the OS will mount as root. The file system will need to comform to the Filesystem Hierachy Standard of Linux, which requires it to have a variuos number of pre-created folders such as /bin, /lib, etc.
2. Bootstraping the OS. This includes boot-loading the kernel, instructing the kernel on how to mount the root file system and what kernel modules to load and where they are located, and providing an init program that the kernel will run after booting has completed.
3. Providing a shell program in which user can enter commands and read outputs.

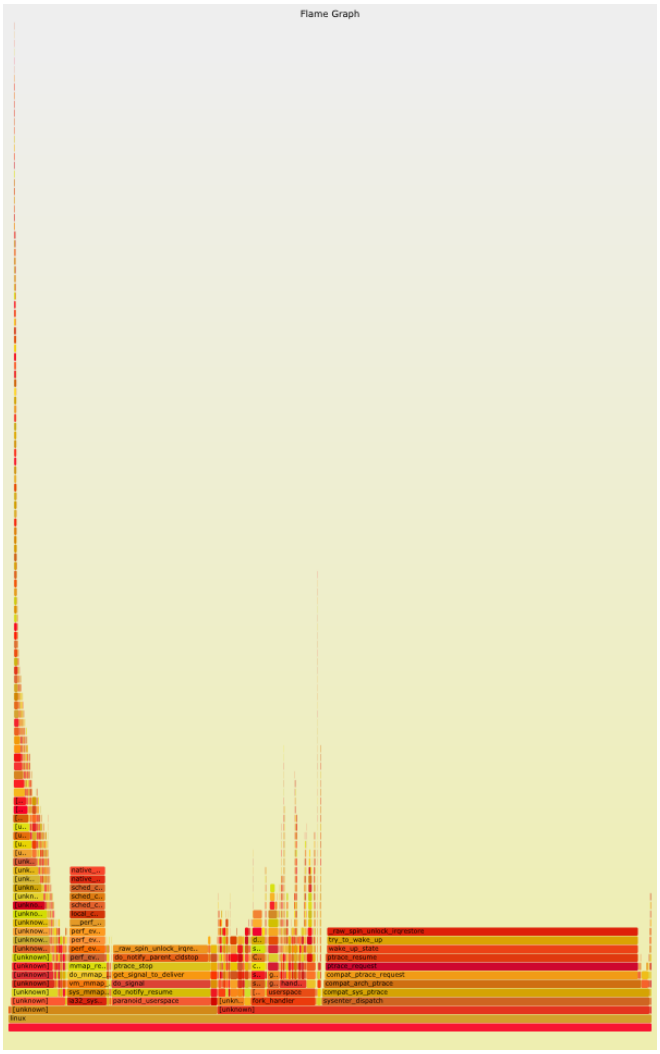

Figure 3: Flame chart for optimized configuration

4. Providing handy tools utilities that users can use to execute common tasks and build softwares, such as a compiler, binary utilities, headers for system APIs, libraries, text editors, manuals and documentations, etc. Though theoretically this part is optional, it is usually highly recommended.

Most popular Linux distributions have configured their system to best meet the needs of an ordinary PC user. On the contrary, we configured MicroswiftOS to best meet the requirements of cloud-hosted applications.

3.2.1 Optimize Boot Scripts

After the kernel boots up, a startup script starts to run to do initialization work, such as setting run levels, mounting file systems, and initializing kernel modules, etc. We can modify some parts in this scripts to speed up the booting process. For example, we can disable the module used for supporting hot-pluggable devices because users will not have physical contacts with the host machine. As a second example, at a certain step the script will wait for udev (a device manager for Linux kernel) initializations to complete before executing the next steps. This is necessary when subsequent initialization steps will use udev; however this is not the case for MicroswfitOS, so we can make subsequent steps to start as soon as possible.

3.2.2 Disable initramfs

initramfs was a mechanism introduced to simplify the booting process. During booting a lot of components such as file systems, disks, device drivers, need to be initialized. However the resouces and configurations that are needed to initialize these components are stored in the file system on disk, therefore it brings the problem of egg and chicken. initramfs is a compressed file system image that includes all essential files needed to initialize the system. If initramfs is enabled, the boot loader will first loaded it directly into the memory, and that the kernel can boot with all essential tools ready. Initramfs simplifies the booting process a lot when the file system or device layout is rather complicated.

We decide to disable initramfs because loading the ramfs into memory takes too much time. Besides, initramfs provides no actual benefit for cloud computing. This is because the disk layout is usually simple for virtual machines, and users are not expected to change any device related configures.

3.2.3 Disable fsck On Boot

Under default settings each partition will need to be checked using fsck before mounted at boot. This is mainly to detect disk corruptions. Since MicroSwiftOS is designed to run in a virtual machine, the disks it sees are actually files residing in the host OS. As a result, fsck would not be necessary because the host file system will be responsible for intergrity checks and providing correct files.

3.2.4 A Minimal Software Collection and a Complete Toolchain

MicroSwiftOS includes everything needed to build a software: a compiler, a linker, a library loader, a text editor, bash utilities, common libraries, debugging tools, manuals and documents. On the other hand, it does not include more software than that. User will need to build and install any softwares that they need to run their application. This will not be a big hassle, because cloud virtual machines will usually be dedicted to only one task, and as a result the virtual machines will only need to be configured once.

4. IMPLEMENTATION

We constructed MicroswiftOS from scratch. The general procedures were similar to what was listed on Linux from Scratch [???], except that a few packages were removed and certain configurations were changed. We choose not to describe every detail as it was a tedious procedure and the exact details can be found online. Instead, we will only highlight general ideas and things we have changed.

4.1 Building a Temporary System

A temporary system will include some basic tools that we need to build every software in MicroswiftOS, like gcc, glibc, binutils, etc. The reason we cannot use tools in the current system is to make sure nothing will be built for the current platform or be linked with libraries in the current system.

4.2 Building Softwares and Tools

Using the tools in the temporary system, we will compile and install every piece of software we need. For a complete list of software installed on MicroswiftOS, check Appendix []

4.3 Buliding the Kernel

The linux kernel is compiled using the configuration detected in the previous section. Besides, the ŞVirtualizationT̈ option must be selected.

4.4 Changing Certain Configuration Files

1. Change the grub configuration file properly so that grub will boot MicroswiftOS.
2. Get rid of the "initrd" line in the grub configuration file to disable ramdisk.
3. Add "loglevel=4" to the kernel commandline arguments so as to decrease the number of messages being logged. This would also slightly speed up the booting process [6].
4. Change /etc/fstab to disable fsck for the root partition.
5. Change /etc/sysconfig/rc.site so as to disable wait for udev_settle.

4.5 Cleanup and Reboot

Remove temporary files used to build system and reboot.

5. EVALUATION

5.1 Environment

We installed MicroswiftOS as a virtual machine instance in Virtualbox 4.3.8, configured to have 4GB virtualized memory and a single processor. It was running on a host computer with processor Intel I3-4130 (3.4GHz, dual core), 8GB physical memory. The host OS was Windows 7. We also installed an instance of Archlinux in Virtualbox as a comparson OS. We have chosen Archlinux as our comparison OS because compared to Gentoo or Ubuntu, it provides a relatively clean and lightweight system after installation.

5.2 System Completeness

In this section we show that MicroswiftOS is able to build and run normal Linux programs correctly with no performance loss. We selected a wide range of workloads to represent the actual cloud computing use cases: thttpd, a selection of some classic algorithms, a benchmark for interprocess communications.

thttpd is an open-source lightweight web server which has a compact code base while delivering superior performance. We have installed thttpd on MicroswiftOS and made it serve a simple HTML page. Then we used another computer to open that web page, and the web page shown was correct. This demonstrates that MicroswiftOS is able to run network-related workloads correctly.

We have also prepared a set of programs that run some classic algorithms: the push-relabel algorithm to solve the maximum flow problem, the Gaussian-Jordan elimination algorithm to solve linear systems, the simplex algorithm to solve linear programmings. We grabbed the code of these algorithms from a code library used for competitive programming contest - this is to say, these implementations have been verified to be correct. We tried to build and run these algorithms on MicroswiftOS under a set of randomly generated input data. Integer arithmetics were covered in the data for max flow, and floating point arithmetics were covered in the data for the linear system solver and the simplex solver. For a detailed description of how input data were generated, see Appendix. We then verified its correctness by comparing its output with the output generated by another computer. No errors were found. These algorithm represents a category of scientific computation workloads, and our test has shown that MicroswiftOS is able to run scientific workloads correctly.

We also benchmarked the performance of inter-process communication on MicroswiftOS, as was conducted in a previous mini-project [???]. Specifically, we measured the latency and throughput of MicroswiftOS, through 3 different inter-process communication mechanisms: pipe, socket, mmap. We also did the same measurement for Archlinux as our control group. Results are shown on Figure []. From the results we learn that the IPC performance of MicroswiftOS and Archlinux are comparable.

5.3 Boot-up Time

We measured the boot-up times of the system under 3 different configurations: an original Archlinux as baseline, ArchlinuxŠs kernel with the image of MicroswiftOS, and MicroswiftOSŠs kernel with MicroswiftOSŠs image. [1] Boot-up times were measured using a stopwatch between the point when a GRUB menu item is selected and the appearance of the login prompt.

Results are shown on Figure []. From the results we know that both our kernel optimizations and image optimizations are effective, with image optimizations being more effective than kernel optimizations, and overall they cut down the boot-up time by half compared to the baseline system.

5.4 System Size

The disk spaces occupied by all MicroswiftOS files were about 900MB, compared to 1.2GB of a newly installed Archlinux. Note that we also included debug symbols of the linux

[1] Originally we also planned to measure the boot time of MicroswiftOSŠs kernel with ArchlinuxŠs image, but it seems that the Archlinux image requires certain kernel configuration options to be selected when compiling the kernel, or otherwise it wonŠt boot. We wasnŠt able to figure it out at the end.

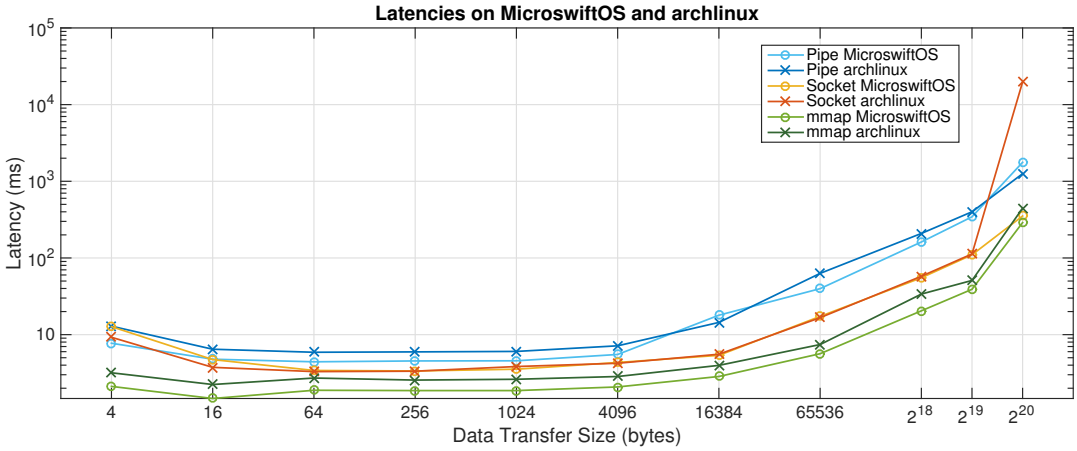

Figure 5: Latencies for various IPC methods on MicroswiftOS and archlinux

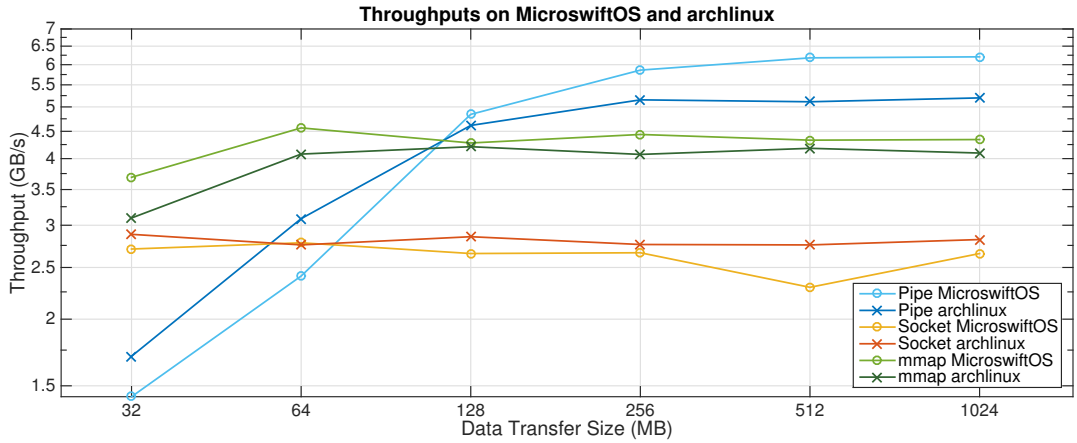

Figure 6: Throughputs for various IPC methods on MicroswiftOS and archlinux

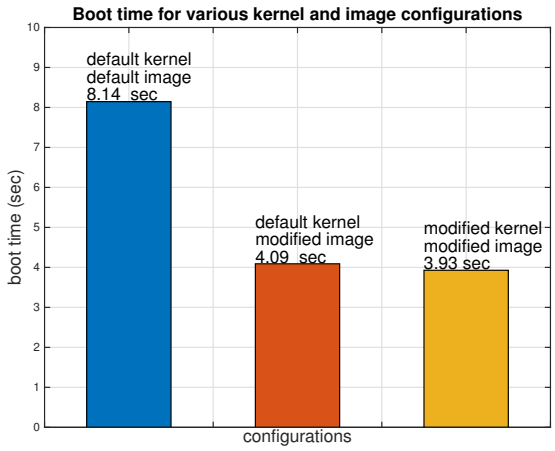

Figure 4: Boot-up time for various configurations

kernel and gcc so as to make diagnosis easier. If debug symbols were removed, the size would be further smaller.

6. RELATED WORK

Kernel design for cloud computing is not a new yet very flourishing field. [5] is a good introductionary survey to cloud computing and it listed 10 big problems in this field. Though many problems have been attempted, more problems are rising over the horizon in the past few years. Kernel minimization does not only draw constant attention from academia, industrial world and computer amateurs also get involved in various communities. Here we list some recent research and projects that inspires our work.

6.1 Minimizing Kernel: Related Research

The Nonkernel, introduced in [4], is a kernel design in which hardware resources are exposed to applications directly to reduce kernel booting time and to boost up parallelism and thus throughput. The nonkernel does three things: 1) machine booting, 2) resource acquisition, and 3) application isolation. Machine booting is an intrinsic requirement of the kernel, and thus should be preserved. Resource acquisition is the management of hardware resources. Specifically, in a cloud computing environment, such mechanism must be provided and the kernel is in charge of resource arbitra-

tion. Finally application isolation is to guarantee that no applications should interfere with each other when accessing the same hardware resources. The nonkernel provides an IPC mechanism that does not involve kernel. It is worth noting that traditional work such as processing scheduling and memory management are no longer the focus of a kernel design in a cloud computing. Nonkernel provides us with reference of cutting down a full-size kernel, yet there is no actual implementation. One major difference is that nonkernel serves as a "supervisor" in cloud computing, while our project aims to build a guest operating system.

Libra [3] is the first operating system that specifically designed for JVM. It borrows the approach similar to exokernel, but avoids the problem of exokernel and reduces the cost of both developing a libOS and administering new partitions. With the use of a hypervisor, applications can use privileged execution modes and instructions, which enables optimizations and eases migration. In our projects we adopted the understanding an OS kernel as many functional parts, and may refer to the approach in this paper.

6.2 Linux Kernel : Methodology, Tools and Tips from Industry

Industrial world and computer amatuers share valuable expereinces on that exploration to reduce boot time. The fastest boot time ever seen is within five seconds [2] measured by Bootchart[1] - Arjan van de Ven managed to boost Fedora 9 on an Asus EEE PC in five seconds, almost 90% reduction in booting time compared to 45 seconds baseline. We find their approach inspiring for our project. Bootchart might be a useful tool for our project, though the last update was back in 2005. Kernel booting consists of three phases: first Readahead module would read blocks of necessary modules from disk so that are cached in the memory, which will reduce cache miss and page fault during booting. Second, it goes along the critical path from filesystem checking to desktop displaying. Third step is hardware abstraction using HAL. These three phases start in sequential order but are running concurrently, and special arrangement has set up to achieve efficient CPU and disk utilization.

To extract basic components in an operating system and get rid of unnecessary ones is no easy task. It usually follows "trial and error" methodology that requires lots of labor and time. In [6], basic OS requirements are defined and elaborated in a cloud computing context. For our project, we used this paper as a reference to set up the minimal feature set for our kernel design.

Mirage in [7] is a specialized software stack designed for cloud use. Mirage is directly on top of the cloud hypervisor, and it entirely replaces the OS, user processes, runtimes, threads, and let applications directly run on Mirage. The advantages of efficiency, security, simplicity and easy deployment come from the characteristic of Mirage that it focus on the domain of I/O intensive cloud servers thus make it possible to specialize the stack. With the lack of OS and other standard software components of a computer, Mirage cannot run general-purpose applications, but can only be used for specialized applications like web servers. On the contrary, what we are trying to build is a general-purpose OS kernel that can run current applications on it without making any extra modifications.

7. FUTURE WORK

8. ACKNOWLEDGEMENT

We would like to thank Michael Swift, Sanketh Nalli and Zhaoyu Luo for their helpful advices and comments.

9. APPENDIX A

A summary of kernel configurations used in MicroswiftOS.

- Kernel hacking All supports in kernel hacking category could be disabled. Once our kernel is sucessfully set up for cloud computing, kernel hacking support could be minimized.
- Library routines All functions and algorithms can be disabled except manadatory ones such as CRC 16 funtions, and CRC32/CRC32c functions.
- Cryptographic API All cryptographic functions could be disabled except manadatory ones such as CSC32c and AES cipher algorithms. The following two are supported by default kernel but could also be eliminated.
  1. Hardware crypto devices
  2. Pseudo Random Number Generation for Cryptographic modules
- Security Options Default security configurations include networking support and cryptographic API. We leave this unchanged as we need to preserve networking related services.
- File System Our operating system uses ext4 file system we disabled support for following file systems.
  1. Reiserfs support
  2. Miscellaneous filesystems
  3. Network file system

  Along with disabled file systems we find two features that could also be disabled.
  1. Native language support
  2. Dnotify support

  Inotify support for userspace could not be removed from the kernel. Experiment shows that external storage devices could not be successfully unlicked and unmounted without inotify support.
- Networking support Default configurations for networking services include sockets, TCP/IP potocol stack. We leave this portion unchanged.
- Device Drivers Block devices and network devices are still supported by default. Sound card support is manatory. IOMMU hardware support could be disabled. Everything else is disabled by default.

- Enable loadable module support This catogory of functionalities can be removed from the kernel. Loadable modules significantly reduce system speed and is not suitable for cloud use.
- General Setup This section includes a lot of configurations which are not tested yet. We have identified the following features that could be removed.
    1. Automatically append version information to the version string
    2.
    3. Support for paging of anonymous memory (swap)
    4. System V IPC
    5. Optimize for size
    6. Enable deprecated sysfs features to support old userspace tools
    7. POSIX message queues
    8. Timer subsystem (all items in it)
    9. Enable access to .config through /proc/config.gz
    10. Kernel .config support
    11. Control group support

    We could disable following features but we recognize that disabling will not necessarily reduce the boot time, hence we leave them as default.

    1. Disable heap randomization
    2. Namespace support
    3. BSD process accounting(in CPU/Task time and stats accounting)

10. APPENDIX B

This section explains how test data were generated. To test the max flow algorithm, we first generate a directed random with 100 vertices, with each possible directed edge having a chance of .1 to appear. Then we associate each edge with a uniformly randomly chosen capacity from the integers in 0..10. At last, two different vertices s (source) and t (sink) are randomly selected. We generated 10 such random networks to test the correctness of the maxflow algorithm.

To test the Gaussian-Jordan elimination algorithm, we generated a 100-by-100 random matrix and a 100-dimensional random vector. Each entry of the matrix and the vector is chosen uniformly at random from real numbers in $[0, 1]$. Floating point tolerence has been set to 10^{-8}, which means two matrices are considered equal if their dimensions match and the absolute difference between each corresponding entry is less than 10^{-8}.

The simplex solver will try to optimize the following problem: maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$, where A, b, c are matrices and vectors of correct dimensions, and x is the solution. To test the correctness of the simplex algorithm, we pick the dimensions of the matrices as follows: A 50-by-30, b 50-by-1, c 30-by-1. All entries in A, b, c are chosen uniformly at random from real numbers in $[0, 1]$. Floating point tolerence has been set to 10^{-8}.

11. REFERENCES

[1] Bootchart. http://www.bootchart.org.

[2] Lpc: booting linux in five seconds. http://lwn.net/Articles/299483/.

[3] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54. ACM, 2007.

[4] Muli Ben-Yehuda, Omer Peleg, Orna Agmon Ben-Yehuda, Igor Smolyar, and Dan Tsafrir. The nonkernel: a kernel designed for the cloud. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 4. ACM, 2013.

[5] Armando Fox, Rean Griffith, A Joseph, R Katz, Andrew Konwinski, Gunho Lee, D Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28:13, 2009.

[6] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. The characteristics of cloud computing. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 275–279. IEEE, 2010.

[7] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud*, volume 10, pages 11–11, 2010.