P

NVIDIA MQP – C Term 2011

Tyler Berg
Patrick Dignan
Sunil Nagpal

March, 24 2011

Table of Contents

List of Figures

Abstract

In the fast moving and high-tech world we live in, more and more systems are being automated for the sake of increased efficiency, reliability, and production quality. The goal of our project was to completely automate the building of new revisions to the source repository, the booting of the entire operating system over the network in a dynamic fashion with software that is not dependent on a specific version of the operating system, and the inclusion of performance and functionality testing into the automatic test infrastructure.

Executive Summary

In the fast moving and high-tech world we live in, more and more systems are being automated for the sake of increased efficiency, reliability, and production quality. The goal of our project was to completely automate the building of new revisions to the source repository, the booting of the entire operating system over the network in a dynamic fashion with software that is not dependent on a specific version of the operating system, and the inclusion of performance and functionality testing into the automatic test infrastructure.

In order to automate the building process, we used Buildbot, a popular tool to automate the testing and building of software. Buildbot is a highly flexible software package, with built in support for numerous source control systems, including subversion, concurrent versions system, git, compilation, and running shell scripts. [13]

NVIDIA needs to make sure their hardware works well with the Chromium OS builds, versions of the software made available in a consumable form. These builds come from compiling the source from the source control system. Since maintaining a high level of quality is extremely difficult, NVIDIA and Google have written a large number of tests for testing the functionality and performance of the hardware and software using autotest to remove the human factor and create consistent tests. However, autotest is not a complete solution, since it does not provide a mechanism for automatically running tests when a new commit is made to the repositories. That's where buildbot comes in. Buildbot supports triggering from various sources, such as periodic triggers, polling, source control commits, HTTP notifications, Gerrit notifications – the system NVIDIA uses to manage code reviews, notifications through its TCP interface, and even emails. The team's implementation of buildbot currently allows for nightly build and test images based on the state of the OS.

One issue that the team faced regarding builds was the the time required to build Chromium OS images. As such, we were asked to build faster build computers, systems designated for compiling the Chromium OS software. Two new computers were built and had the development environment installed on them, while a third computer needed RAM before more could be done with it. The fastest of these new computers cut the total build time from about 6 hours to about 3.5 hours, removing a huge bottleneck for the team. In addition, after the completion of our MQP, these computers will serve as build computers for the development team, since some of the current build computers take up to 12 hours to complete one build. Building the computers was largely a process of debugging broken and incompatible parts as well as locating appropriate hardware.

After automating the build process, we were tasked with finding a way to automate the rollout of new filesystem images onto the test boards on a nightly basis. Typically, one would access the local shares and load a nightly build/image onto a USB stick, and then mount this image

onto either the harmony or seaboard. By automating this process, we would eliminate the need for this manual step.

We chose to use the “NFS root” method, which allows one’s target machine (the test board) to use a NFS mount on your host Linux machine as its root filesystem (rootfs). This is a much faster process than loading the changes onto a USB device and rebooting the target machine using that filesystem. In addition, since the filesystem is stored on a remote host, space is rarely, if ever, a concern, since it is simple to add more storage to a server or workstation. This is particularly useful when working with debug images, which are typically much larger than standard images. [17]

Enabling NFS boot to work with our host and target machines takes away the need to manually load nightly images onto our test boards, and allows a developer to utilize the large amount of disc space available on the host machines.

To further automate the process, we tested the use of DHCP (Dynamic Host Configuration Protocol) Boot. DHCP Boot is very similar to NFS Boot because it uses a mounted root filesystem that is not pre-loaded onto the test board. It further improves on NFS Boot as a means of automation, though, because it loads the system kernel over a DHCP/TFTP server rather than from a USB device attached to the test board. Using DHCP Boot allows a user to load both the kernel and root filesystem over the network. [18]

The configuration for DHCP Boot does not require many more steps than NFS Boot. First a user must set up a DHCP Server and a TFTP (Trivial File Transfer Protocol) Server. The DHCP Server is used to provide IP addresses to targets on the network, and the TFTP Server is used to send the kernel to U-Boot when it requests one. After setting up these servers, a user needs to define some boot arguments such as the server IP addresses, specific bootfiles, and a path to the kernel on the host machine. With this setup, a user can boot their test board with the kernel and root filesystem completely loaded over a local network.

After automating the boot process, we worked on automating testing. NVIDIA currently has a large suite of proprietary tests, called the nvtest suite, which are run on their development boards to ensure the software is running properly. These proprietary tests are currently run one-by-one by testers in NVIDIA’s Quality Assurance group. Our group was tasked with automating these tests. Chromium OS uses a test harness called autotest to automatically run its test suite. Autotest allows tests to be run on remote devices in a repeatable, parallelizable, and efficient way. Automating these tests frees up Quality Assurance resources for better and more in-depth testing. In addition, automating these tests allows developers to get more rapid feedback to their changes. Faster feedback means that issues can be found before the developer merges their changes with the main tree, preventing their bugs from propagating to the rest of the developers.

First we had to figure out how to run NVIDIA’s proprietary tests, and then write the adaptor that allows these tests to be run by autotest. Once we figured out how to run a few tests, it became clear that automating creation of the adaptors would benefit the team greatly. In order to accomplish our deliverable of having a working nvtest running through autotest, the team created a program that generates the adaptor. The generator was created successfully and allowed the team to rapidly port NVIDIA's proprietary tests to autotest.

Autotest allows tests to be run on computers connected over the network. In this regard, autotest has a client-server architecture. There is a target board, which is the hardware upon which the tests are to be run, and a test server, typically a developer's computer which sends the tests over the network to the target board. We realized that this architecture required that the target board be online to run the nvtests through autotest, so that the test server could access the target board. After this was done the team used the scripts nvidia has written for using autotest to run the autotests we had created. This works correctly and now testing can be done from several computers onto one board simultaneously.

Acknowledgements

We would like to acknowledge those who have helped us in the successful completion of our project. First and foremost, we appreciate the efforts of Professor Finkel, who enabled us to have an excellent project to work on in the first place through his diligence. Further, Professor Finkel has been a guiding force throughout the entirety of our project, providing useful critiques as the project passed. Next we would like to thank Larry Robinson, Allen Martin, and Matt Pedro, without whom we would not have had a project. They have provided great oversight of our project, and enabled us to quickly integrate into NVIDIA so that we were able to get our work done. We would like to thank the rest of the team for welcoming us into the team, helping us when we ran into roadblocks in our project, and making our time at NVIDIA enjoyable. WPI has our eternal gratitude for providing us with the education we needed to complete the work on our project and setting up the MQP program. Finally, we would like to thank Simon Glass, from Google, who provided expert insight into several issues the team ran into, and for showing us around Google.

1.0 Introduction

When we began doing research for our project in November of 2010, we found that our project description was not very specific. We were unsure what topics to research because we were only told that we would be working on performance measuring, Google's Chrome OS, and the Tegra Chipset. The focus of the project, being somewhat opaque, led us to look in depth at the topics presented above. As such, our Background chapter includes our research on the Tegra chipset, Chrome OS, and several performance-monitoring tools on various operating systems.

When we arrived at NVIDIA in January of 2011, though, the focus of our project became much clearer. We determined that the goal of our project would be to completely automate the building of new revisions to the source repository, the booting of the entire operating system over the network in a dynamic fashion with software that is not dependent on a specific version of the operating system, and the inclusion of performance and functionality testing into the automatic test infrastructure.

Our project was focused around Chromium OS, Google's open source project which aims to develop a browser-based operating system. Since Chromium OS is open source and still under development, there are new versions of the operating system put out almost daily. We were tasked with keeping NVIDIA's builds of Chromium OS up to date with Google's by automating the build phase. Next we were tasked with being able to provide a reliable method for loading the operating system over a network rather than off of test system hardware in order to speed up debugging time and maximize the disk space available to a test machine. Finally, we worked on including several performance and functionality testing capabilities into the automatic test infrastructure.

After nine weeks of work, we successfully completed our project. All goals were met and additional work was done besides what was planned at the offset of the project in January.

2.0 Background

2.1 NVIDIA; About the Company

NVIDIA was founded in January 1993 by Jen-Hsun Huang, former director of coreware at LSI Logic Corp, Chris Malachosky, and Curtis Priem. Beginning as a company fueled by just intellectual property, NVIDIA has grown to be a leader in everything including graphics processing units, integrated circuits, and chipsets. One dynamic that makes NVIDIA unique from its competitors is its range of platforms on which the company's products not only survive, but thrive. NVIDIA has their products in computers, gaming consoles, phones, and tablets. In 2007, NVIDIA was named Company of the Year by Forbes magazine. [1]

2.2 Business Background

In 1995 NVIDIA released its first product known as the NV1, a multimedia PCI card. Manufactured by NVIDIA's first acquisition, SGS-THOMSON Microelectronics, the card received mixed reviews in that it was expensive for its quality and the quadratic rendering was different from the mainstream polygon rendering. Not rendering polygons meant that the card did not support Direct3D but in 1997 NVIDIA created the RIVA 128, a card meant to push Direct3d to its furthest capabilities. With the release of the RIVA 128, NVIDIA started to push its way out of the shadows of competitors and into multimedia fame. Two years later the company went public and sold its ten millionth graphics processor. A year after that the company absorbed one of the largest graphics companies of the 90's. That same year Microsoft offered NVIDIA a contract to supply GPUs for their next generation gaming console, the X-Box. NVIDIA has since discontinued their service to Microsoft gaming consoles, and is to this date supplying Sony GPU's for the PlayStation 3. NVIDIA also makes integrated circuits for mobile devices called System-on-a-chip which contains a CPU, GPU, northbridge, southbridge, memory controller, and the ARM architecture. NVIDIA calls this series TEGRA. [2]

2.3 Tegra Chipset

NVIDIA's Tegra series chipsets contains some of the world's most powerful mobile processors to date. [5] The Tegra chipset contains an ARM architecture processor CPU, GPU, northbridge, southbridge, and memory controller. The northbridge is used for memory controlling while the southbridge controls input and output logic. Tegra chips range from 600 MHz to 1 GHz processing speed, and use low power DDR and DDR2 memory. The Tegra chipset is used today in smart phones and tablet PC's. [4]

2.4 Chrome OS

Chrome OS is Google's challenge to enter the operating system market. Based on Linux, the primary goals and features are improved boot speeds, better security, more simplicity, and integration with web services. They plan to achieve this by making the browser the only program to run on the computer. In addition to this, modified firmware would allow Chrome OS to have certain security features that are not present in other operating systems, such as firmware, which can verify the integrity of the operating system kernel, which can then verify the rest of the operating system. Google also plans to streamline hardware detection using the special firmware. [3]

Chrome OS is a rebranded version of an open source project, Chromium OS. This is a parallel to the relationship between Google's Chrome web browser and the Chromium project. Chrome OS will require specific hardware in order to take advantage of Google's optimizations for performance and security.

2.5 Windows Performance Tools

2.5.1 Task Manager

Task Manager is Windows' most widely known performance tool. It allows the user to examine and control processes and services as well as monitor the performance and networking connections associated with the system. Although it is a powerful tool, and even given a shortcut key command in Windows, it is still very limited in its scope. It presents individual process information only for the timeframe that the user is viewing task manager. Figure 1 shows Windows Task Manager and the memory and the CPU usage of some of the processes.

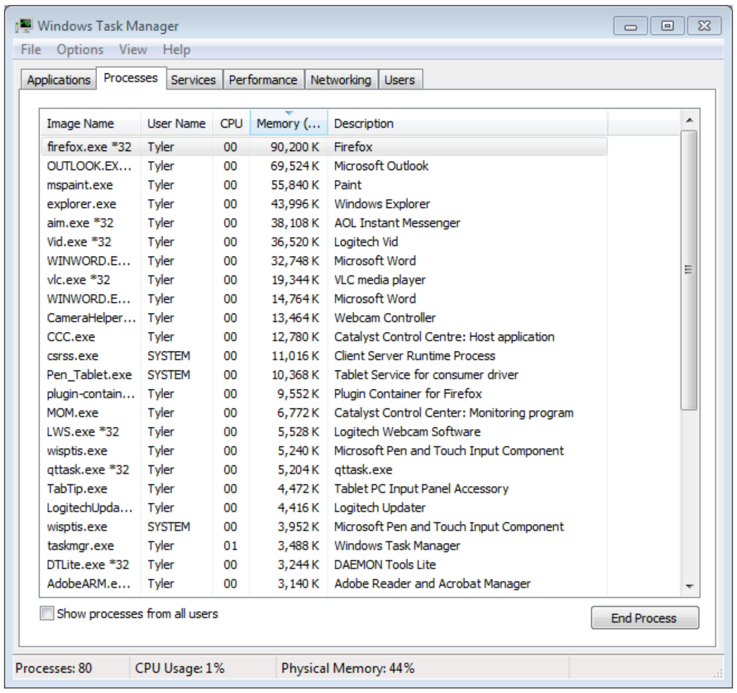

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

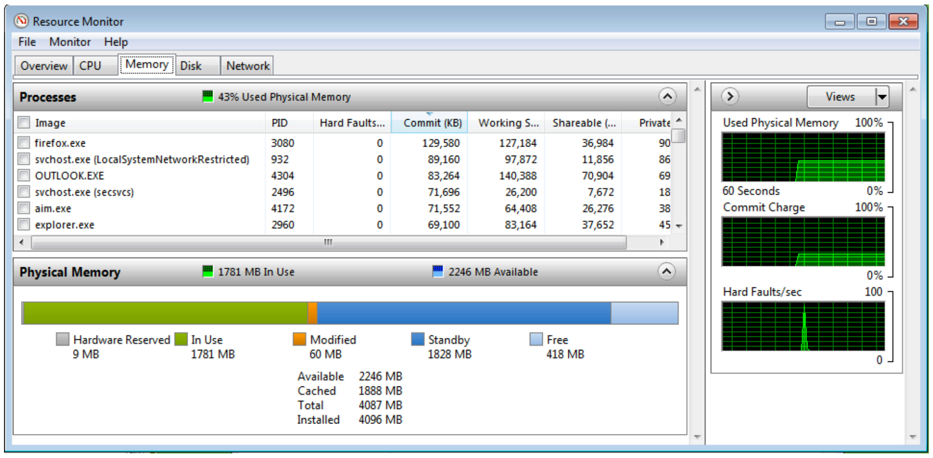
Image Name	User Name	CPU	Memory (...	Description
firefox.exe *32	Tyler	00	90,200 K	Firefox
OUTLOOK.EX...	Tyler	00	69,524 K	Microsoft Outlook
mspaint.exe	Tyler	00	55,840 K	Paint
explorer.exe	Tyler	00	43,996 K	Windows Explorer
aim.exe *32	Tyler	00	38,108 K	AOL Instant Messenger
Vid.exe *32	Tyler	00	36,520 K	Logitech Vid
WINWORD.E...	Tyler	00	32,748 K	Microsoft Word
vlc.exe *32	Tyler	00	19,344 K	VLC media player
WINWORD.E...	Tyler	00	14,764 K	Microsoft Word
CameraHelper...	Tyler	00	13,464 K	Webcam Controller
CCC.exe	Tyler	00	12,780 K	Catalyst Control Centre: Host application
csrss.exe	SYSTEM	00	11,016 K	Client Server Runtime Process
Pen_Tablet.exe	SYSTEM	00	10,368 K	Tablet Service for consumer driver
plugin-contain...	Tyler	00	9,552 K	Plugin Container for Firefox
MOM.exe	Tyler	00	6,772 K	Catalyst Control Center: Monitoring program
LWS.exe *32	Tyler	00	5,528 K	Logitech Webcam Software
wisptis.exe	Tyler	00	5,240 K	Microsoft Pen and Touch Input Component
qttask.exe *32	Tyler	00	5,204 K	qttask.exe
TabTip.exe	Tyler	00	4,472 K	Tablet PC Input Panel Accessory
LogitechUpda...	Tyler	00	4,416 K	Logitech Updater
wisptis.exe	SYSTEM	00	3,952 K	Microsoft Pen and Touch Input Component
taskmgr.exe	Tyler	01	3,488 K	Windows Task Manager
DTLite.exe *32	Tyler	00	3,244 K	DAEMON Tools Lite
AdobeARM.e...	Tyler	00	3,140 K	Adobe Reader and Acrobat Manager

Processes: 80 CPU Usage: 1% Physical Memory: 44%

Figure 1: Windows Task Manager

2.5.2 Resource Monitor

The Resmon tool, which is present in all Windows OS's since Windows Vista, is used to monitor computers' resources. Like Windows Task Manager, Resmon is able to look at, pause, and end applications and their associated processes. It can also evaluate and present data about each of the processes from how much memory the process is using to how many threads the process contains. Unlike Windows Task Manager, Resmon can calculate the average CPU that each process uses, and can even show processes that have already been terminated. Resmon's scope includes CPU, memory, disk, and network usage information. [6] Figure 2 shows the resmon.exe application. On the top are five tabs (Overview, CPU, Memory, Disk, Network) that display information about that piece of hardware and how it is being utilized.



Processes — 43% Used Physical Memory

Image	PID	Hard Faults...	Commit (KB)	Working S...	Shareable (...	Private
firefox.exe	3080	0	129,580	127,184	36,984	90
svchost.exe (LocalSystemNetworkRestricted)	932	0	89,160	97,872	11,856	86
OUTLOOK.EXE	4304	0	83,264	140,388	70,904	69
svchost.exe (secsvcs)	2496	0	71,696	26,200	7,672	18
aim.exe	4172	0	71,552	64,408	26,276	38
explorer.exe	2960	0	69,100	83,164	37,652	45

Physical Memory — 1781 MB In Use — 2246 MB Available

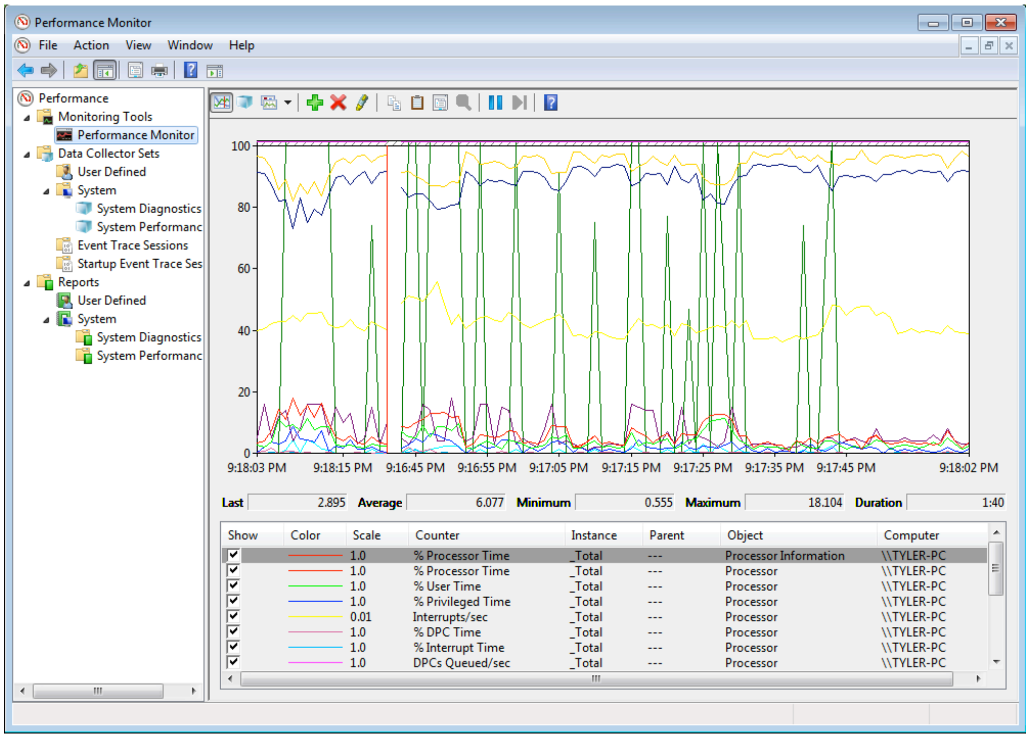
Hardware Reserved	In Use	Modified	Standby	Free
9 MB	1781 MB	60 MB	1828 MB	418 MB

Available	2246 MB
Cached	1888 MB
Total	4087 MB
Installed	4096 MB

Figure 2: Resource Monitor - resmon.exe

2.5.3 Performance Monitor

Perfmon.exe is a performance-monitoring tool in the more recent versions of Windows. It differs from Windows Task Manager and Resmon in that it does not have control of specific processes. It does however, have the ability to measure performance over several hundred metrics. Perfmon also allows the saving and comparison of these metrics over any time period. Figure 3 shows tracking of several processing metrics over the time of about two minutes. Perfmon also displays the average, maximum, and minimum of the selected metric, or metrics.



Last	Average	Minimum	Maximum	Duration
2.895	6.077	0.555	18.104	1:40

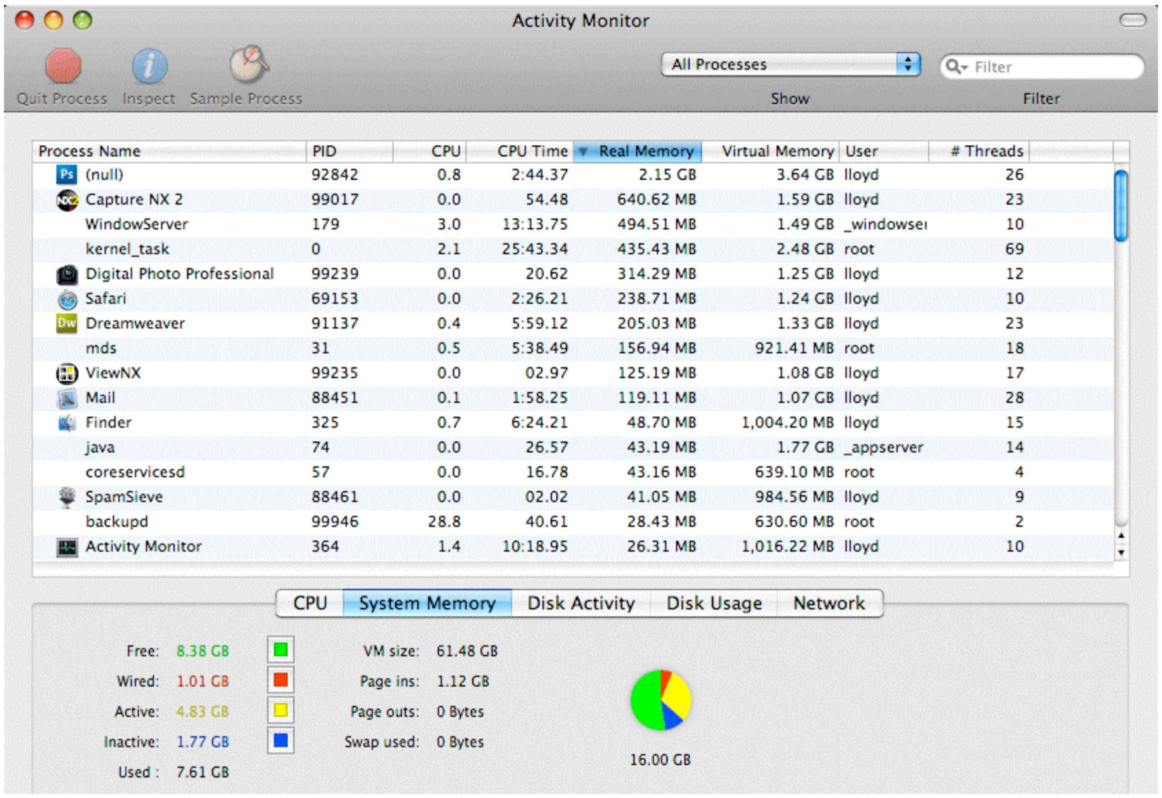
Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	% Processor Time	_Total	---	Processor Information	\\TYLER-PC
✓	🟥	1.0	% Processor Time	_Total	---	Processor	\\TYLER-PC
✓	🟩	1.0	% User Time	_Total	---	Processor	\\TYLER-PC
✓	🟦	1.0	% Privileged Time	_Total	---	Processor	\\TYLER-PC
✓	🟨	0.01	Interrupts/sec	_Total	---	Processor	\\TYLER-PC
✓	🟪	1.0	% DPC Time	_Total	---	Processor	\\TYLER-PC
✓	🟦	1.0	% Interrupt Time	_Total	---	Processor	\\TYLER-PC
✓	🟪	1.0	DPCs Queued/sec	_Total	---	Processor	\\TYLER-PC

Figure 3: Performance Monitor - perfmon.exe

2.6 Mac OS – Activity Monitor/Disk Utility – System Performance Tools

2.6.1 Activity Monitor

In the Mac OS, a user can observe their system performance using the built-in Activity Monitor. The Activity Monitor can be used to check the memory usage, CPU usage, disk activity, battery health and many other metrics of system performance. When a user opens the Activity Monitor, the window appears as in Figure 4. [7][8]



Show: All Processes

Process Name	PID	CPU	CPU Time	Real Memory	Virtual Memory	User	# Threads
(null)	92842	0.8	2:44.37	2.15 GB	3.64 GB	lloyd	26
Capture NX 2	99017	0.0	54.48	640.62 MB	1.59 GB	lloyd	23
WindowServer	179	3.0	13:13.75	494.51 MB	1.49 GB	_windowsei	10
kernel_task	0	2.1	25:43.34	435.43 MB	2.48 GB	root	69
Digital Photo Professional	99239	0.0	20.62	314.29 MB	1.25 GB	lloyd	12
Safari	69153	0.0	2:26.21	238.71 MB	1.24 GB	lloyd	10
Dreamweaver	91137	0.4	5:59.12	205.03 MB	1.33 GB	lloyd	23
mds	31	0.5	5:38.49	156.94 MB	921.41 MB	root	18
ViewNX	99235	0.0	02.97	125.19 MB	1.08 GB	lloyd	17
Mail	88451	0.1	1:58.25	119.11 MB	1.07 GB	lloyd	28
Finder	325	0.7	6:24.21	48.70 MB	1,004.20 MB	lloyd	15
java	74	0.0	26.57	43.19 MB	1.77 GB	_appserver	14
coreservicesd	57	0.0	16.78	43.16 MB	639.10 MB	root	4
SpamSieve	88461	0.0	02.02	41.05 MB	984.56 MB	lloyd	9
backupd	99946	28.8	40.61	28.43 MB	630.60 MB	root	2
Activity Monitor	364	1.4	10:18.95	26.31 MB	1,016.22 MB	lloyd	10

Free:	8.38 GB	🟩	VM size:	61.48 GB
Wired:	1.01 GB	🟥	Page ins:	1.12 GB
Active:	4.83 GB	🟨	Page outs:	0 Bytes
Inactive:	1.77 GB	🟦	Swap used:	0 Bytes
Used :	7.61 GB			

16.00 GB

Figure 4: Activity Monitor [8]

This main window allows one to view all active processes, as well as the available and used CPU, system memory, and disk resources. Each item is accessible through the tabs in the lower portion of the window. By navigating through each tab, a user can learn about how the computer's resources are being used. Figure 5 shows how one can monitor the recent CPU usage. This is a helpful feature because it provides a live graph that differentiates between user-run and system-run processes, with each indicated by a different color. [7]

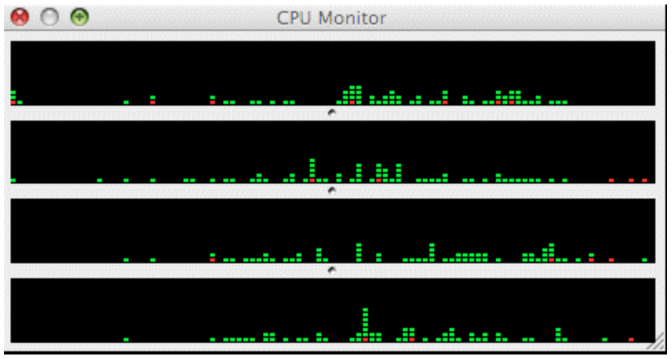
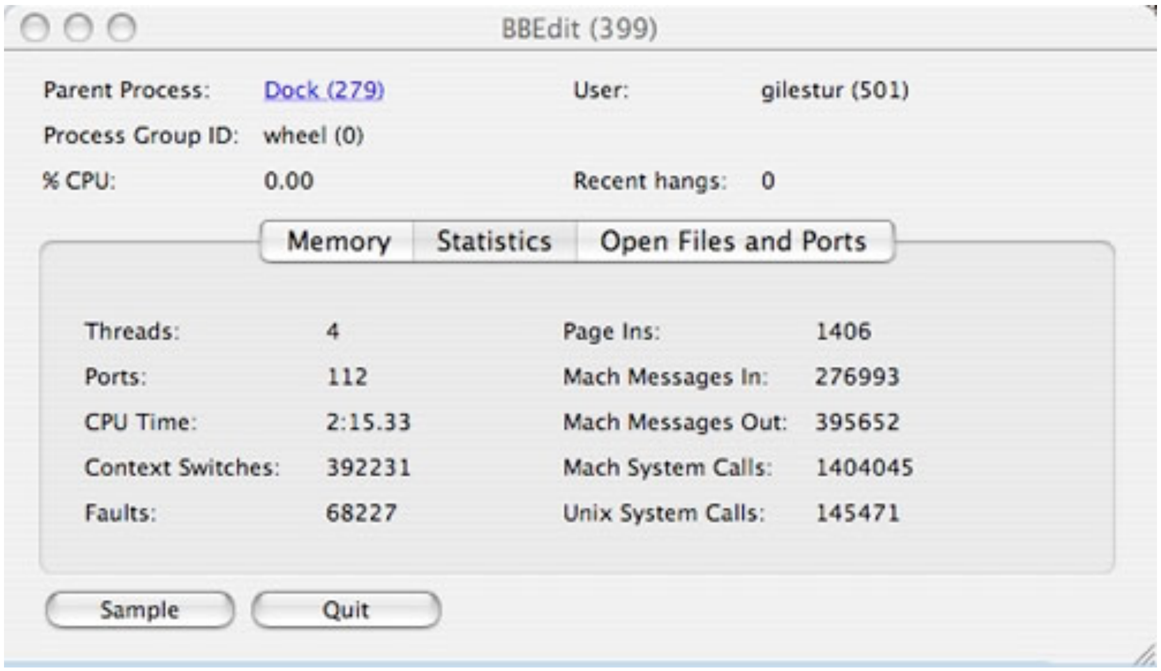


Figure 5: CPU Monitor

In addition, one has the ability to analyze any individual process in more detail. A user can open an information panel for a process, as shown below in Figure 6, which details the number of active threads, ports, faults, system calls, etc. If desired, a user can dock the monitoring graphs on their desktop while doing other work. The easy access to system performance tools is a helpful feature that enables multitasking while observing the system resources. [8]



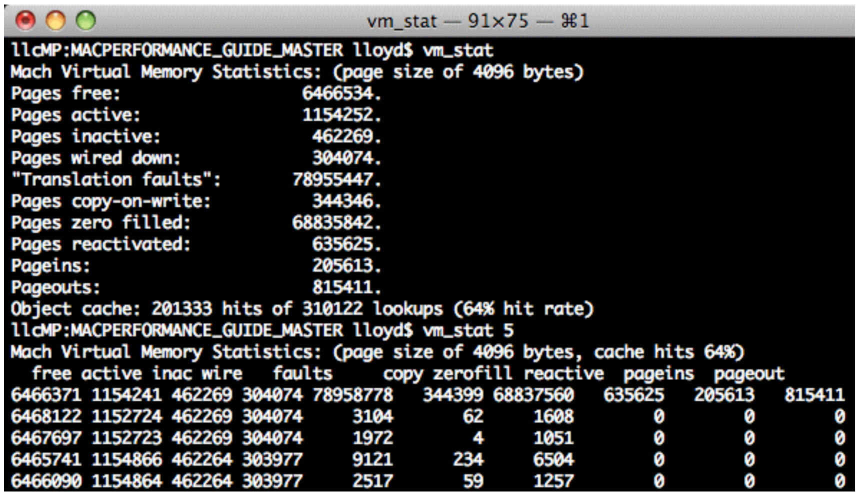
Parent Process:	Dock (279)	User:	gilestur (501)
Process Group ID:	wheel (0)		
% CPU:	0.00	Recent hangs:	0

Threads:	4	Page Ins:	1406
Ports:	112	Mach Messages In:	276993
CPU Time:	2:15.33	Mach Messages Out:	395652
Context Switches:	392231	Mach System Calls:	1404045
Faults:	68227	Unix System Calls:	145471

Figure 6: BBedit

2.6.2 Vmstat

An experienced user, however, might prefer the command-line interface that is also available in the Mac OS. The Linux based vm_stat command, when run in a Terminal (command prompt), lets a user view the same system information in a simple table, as shown in Figure 7. [8]



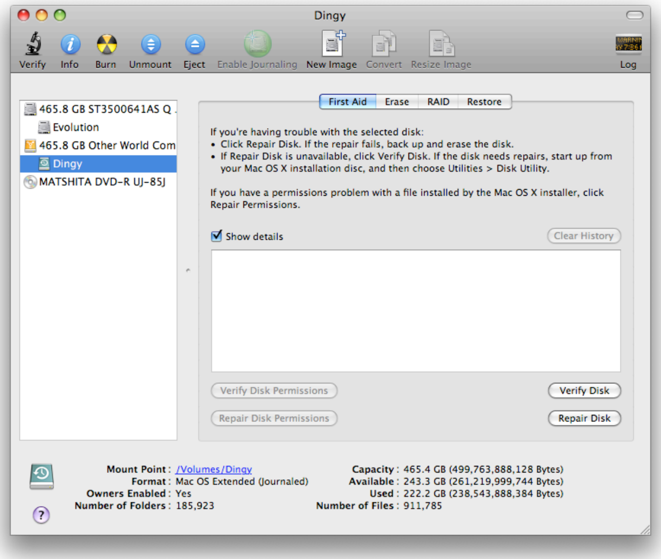
```
llcMP:MACPERFORMANCE_GUIDE_MASTER lloyd$ vm_stat
Mach Virtual Memory Statistics: (page size of 4096 bytes)
Pages free:                   6466534.
Pages active:                 1154252.
Pages inactive:                462269.
Pages wired down:              304074.
"Translation faults":        78955447.
Pages copy-on-write:           344346.
Pages zero filled:           68835842.
Pages reactivated:             635625.
Pageins:                       205613.
Pageouts:                      815411.
Object cache: 201333 hits of 310122 lookups (64% hit rate)
llcMP:MACPERFORMANCE_GUIDE_MASTER lloyd$ vm_stat 5
Mach Virtual Memory Statistics: (page size of 4096 bytes, cache hits 64%)
  free active inac wire   faults     copy zerofill reactive  pageins  pageout
6466371 1154241 462269 304074 78958778   344399 68837560   635625   205613   815411
6468122 1152724 462269 304074     3104       62     1608        0        0        0
6467697 1152723 462269 304074     1972        4     1051        0        0        0
6465741 1154866 462264 303977     9121      234     6504        0        0        0
6466090 1154864 462264 303977     2517       59     1257        0        0        0
```

Figure 7: Terminal Run System Performance Tools [7]

Each of these tools is helpful in providing a user with live system performance information without using up much of the system resources themselves.

2.6.3 Disk Utility

Another tool that enables a Mac user to monitor their system resources and performance is the Disk Utility. The Utility allows a user to mount disks (both hard disks and virtual images), format disk drives, analyze the health of active disks and provide repairs if necessary, as well as several other features. The Disk Utility, which can also be accessed by the Terminal commands *diskutil* and *hdiutil*, is shown in Figure 8. [7]



If you're having trouble with the selected disk:
- Click Repair Disk. If the repair fails, back up and erase the disk.
- If Repair Disk is unavailable, click Verify Disk. If the disk needs repairs, start up from your Mac OS X installation disc, and then choose Utilities > Disk Utility.

If you have a permissions problem with a file installed by the Mac OS X installer, click Repair Permissions.

☑ Show details

Mount Point : /Volumes/Dingy
Format : Mac OS Extended (Journaled)
Owners Enabled : Yes
Number of Folders : 185,923

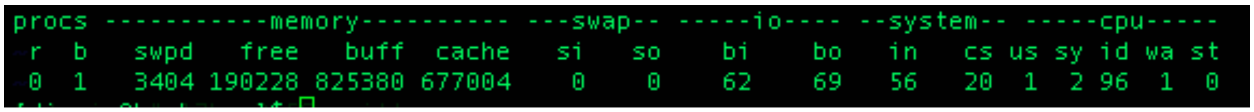
Capacity : 465.4 GB (499,763,888,128 Bytes)
Available : 243.3 GB (261,219,999,744 Bytes)
Used : 222.2 GB (238,543,888,384 Bytes)
Number of Files : 911,785

Figure 8: Disk Utility [7]

2.7 Linux Performance Tools

2.7.1 Vmstat

Due to Linux's fragmented nature, there are myriad performance tools. For the sake of this paper, vmstat will be discussed. Vmstat provides performance information regarding memory and CPU usage. Unlike some of the other monitoring tools mentioned, vmstat is not capable of interfering with processes or memory; it only displays information about them. The main advantage to vmstat is the simplicity of the interface. For example, disk information can be viewed simply by typing the vmstat -d command. This will tell the user information including the number of past read/writes as well as current IO information. Further, the user can drill down into specific partitions using the vmstat -d <partition> command [9]. In running vmstat, the user can view information regarding RAM, swap, IO, system, and CPU usage.



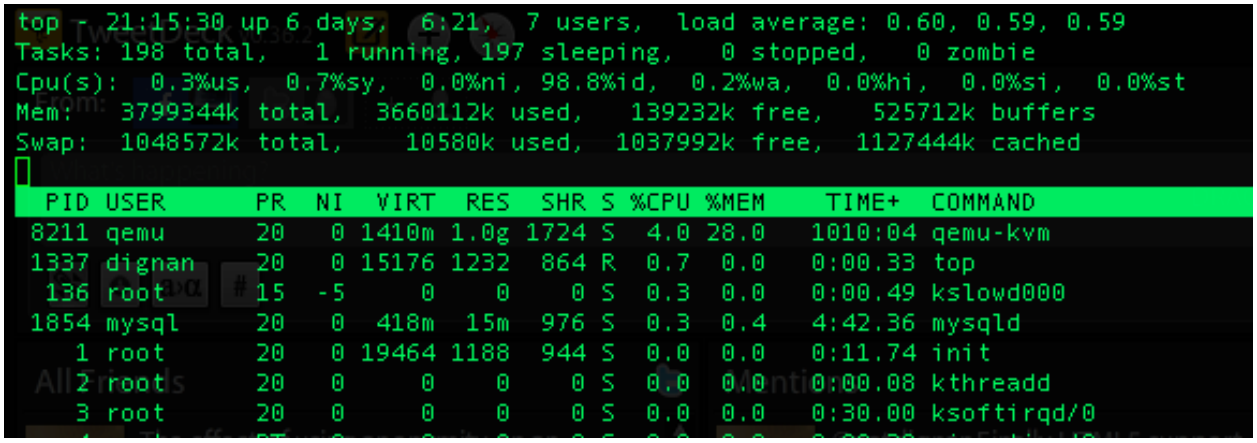
```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  1   3404 190228 825380 677004    0    0    62    69   56   20  1  2 96  1  0
```

Figure 9: Vmstat

Figure 9 shows the output of running the vmstat command without any parameters. The top row shows the general headings, procs for processes and io for input and output. The second row tells what specific metric is in the third row. The r column says how many processes are waiting for runtime, and the b column says how many processes are in an uninterrupted sleep state. The swpd column says how much virtual memory is used, the free column says how much is idle, the buff column says how much is in use as buffers, while the cache column shows how much memory is in use as cache. For the swap column, si and so show how much memory is swapped in from the disk per second and how much memory is swapped out to the disk per second, respectively. In the io column, the bi and bo columns show how many blocks are swapped in and out of the disks per second. The system column shows the number of interrupts per second and the number of context switches per second in the in and cs columns, respectively. The last column contains relevant CPU information. There are 5 columns, representing different categories of CPU usage as percentages. The us column shows how much time is used by user processes, the sy column shows how much time is used by the kernel, the id column shows the amount of CPU spends idle, the wa column shows the amount of time spent waiting for IO systems, and the st column shows the amount of time stolen from a virtual machine. [10]

2.7.2 Top

Another important performance monitoring tool in Linux is the top command. Top is one of the first commands a new user of the Linux command line learns. By running top, the user gets a list of the active processes sorted by usage that is updated constantly until the user exits by pressing “q”. Having several criteria on which to be sorted, the top command is considered by some, to be very flexible. In addition, top has a summary of CPU, process, and RAM information at the top of the display. Top is useful for identifying processes that use large chunks of CPU as well as for revealing the id’s of those processes. This in turn can be used to shut those processes down.



```
top - 21:15:30 up 6 days,  6:21,  7 users,  load average: 0.60, 0.59, 0.59
Tasks: 198 total,   1 running, 197 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3%us,  0.7%sy,  0.0%ni, 98.8%id,  0.2%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3799344k total,  3660112k used,   139232k free,   525712k buffers
Swap:  1048572k total,    10580k used,  1037992k free,  1127444k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8211 qemu      20   0 1410m 1.0g 1724 S  4.0 28.0   1010:04 qemu-kvm
 1337 dignan    20   0 15176 1232  864 R  0.7  0.0   0:00.33 top
  136 root      15  -5     0    0    0 S  0.3  0.0   0:00.49 kslowd000
 1854 mysql     20   0  418m  15m  976 S  0.3  0.4   4:42.36 mysqld
    1 root      20   0 19464 1188  944 S  0.0  0.0   0:11.74 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.08 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:30.00 ksoftirqd/0
```

Figure 10: Top in Action

The screenshot in Figure 10 shows the top command running on a Fedora 14 server. The top row tells the time, the uptime of the computer, the number of users active, and the load average. The load numbers indicate processor usage averages over 1, 5, and 15 minute periods, respectively. The load represents the percentage of a CPU that the processes are using. For example, the 0.60 indicates that the load is enough to use 60% of one of the processors on the computer. Fortunately, this computer has 4 cores, so that means the load is only enough to saturate 60% of one of the four cores. [11]

The next line in Figure 10 shows the number of processes and some other straightforward information regarding their state. The CPU line should look familiar from vmstat's output, and the next two lines are fairly self-explanatory. The more interesting functionality provided by top is in the table below the header. The first column is the PID, the process ID number which is a unique identifying number assigned to a process that can be used to end it, trace it, and many other things. The user column shows which user is executing the task, while the PR column shows what the processes priority is. The NI column is similar to the PR column, but tells what the nice value is. Nice allows the user to change the priority of a process. The VIRT column shows how much virtual memory is used by the task. The RES column tells how much unswapped physical memory the task is using. The SHR column shows how much memory is used that could potentially be shared with other processes. The S column shows what the state of the task is, the D means uninterruptible sleep, the R means running, the S means sleeping, the T means traced or stopped, and the Z means zombie. The last three columns are fairly self-explanatory as well. [12]

3.0 Process/Results

3.1 The Overarching Goal of our Project

The overarching goal of our project was to completely automate the build- the creation of a consumable form of Chromium OS, booting from a remote server, and finally testing the build. The work we did on the build system will allow NVIDIA to automatically create builds, compiled versions of the Chromium OS software, for new revisions to the source repository, the server that hosts the version controlled software source code. In order to test the build that was automatically generated, it needs to be installed onto hardware for performance and functionality testing. Currently, the performance and functionality tests need to be done manually, but we have tested and documented a process for booting the entire operating system over the network in a dynamic fashion with software that is not dependent on a specific version of the operating system, since the operating system is constantly under development, and there are new versions all the time.

At the outset of the project, the automation of testing was somewhat haphazard. Functionality tests had been made into a form such that they could be run from the command line, but had no way to fit into the existing infrastructure and were run by hand. NVIDIA uses buildbot [13], an automation platform, for their testing infrastructure. Performance testing was not automated at all, and was performed by going to websites and manually running the tests. Our project automated the browser-based performance tests, such as Sunspider and Page Cycler, which test JavaScript execution speed and page rendering speed, and fit both the performance and functionality testing into the automatic test infrastructure. Functionality testing includes things such as ensuring that the video player works on the test hardware. Using the aforementioned build and boot techniques and combining them with the automated testing provides a powerfully automated development framework that will improve code and shorten the feedback cycle for NVIDIA's developers when they are developing Chromium OS.

3.2 Buildbot

One of the most important aspects of working on a large software project is automation. Testing needs to be automated, building needs to be automated, and anything that can possibly be automated should be. Buildbot is a popular tool to automate the testing and building of software. Buildbot is a highly flexible software package, with built in support for numerous source control systems, including subversion, concurrent versions system, git, compilation, and running shell scripts. [13]

Chromium OS is a complicated software project. NVIDIA needs to make sure their hardware works well with the Chromium OS builds, versions of the software made available in a consumable form. These builds come from compiling the source from the source control system. Builds are available from the chromium.org buildbot or internally at NVIDIA from a shared network drive. Since maintaining a high level of quality is extremely difficult, NVIDIA and Google have written a large number of tests for testing the functionality and performance of the hardware and software using autotest to remove the human factor and create consistent tests. However, autotest is not a complete solution, since it does not provide a mechanism for automatically running tests when a new commit is made to the repositories. A commit is a set of code submitted to a repository, the version-controlled area where code is stored. That's where buildbot comes in. Buildbot supports triggering from various sources, such as periodic triggers, polling, source control commits, HTTP notifications, Gerrit notifications – the system NVIDIA uses to manage code reviews, notifications through its TCP interface, and even emails. The mobile team's buildbot instance currently builds and tests nightly images based on the state of the OS.

Buildbot is broken into two major components which essentially makes up a client-server architecture. There is a master which controls scheduling, interfacing with the web, and listening for changes from the various change sources mentioned before. The second component part is the slave. Slaves are the components that actually execute the instructions for which the instance is configured. The client-server architecture employed by buildbot allows for a many to one relationship of slaves to masters. As a technical side-note: you can actually have backup masters.

Buildbot configuration is very straightforward. Thanks to buildbots decision to use the programming language Python, a language designed for readability, in its configuration files. This keeps the configuration powerful and simple. Configuration of slaves is mostly done by creating a folder and running the buildslave script:

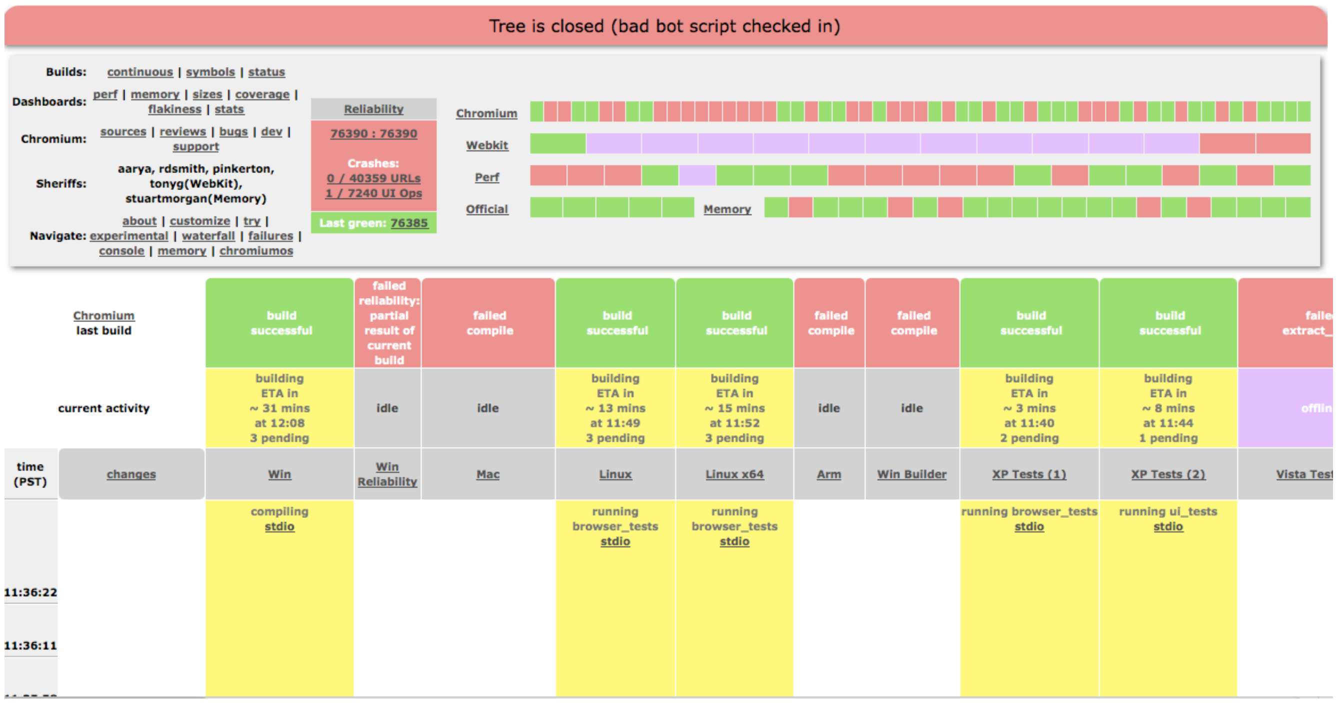
```
buildslave create-slave BASEDIR MASTERHOST:PORT SLAVENAME PASSWORD
```

BASEDIR should be whatever directory you want the buildslave to be based out of, while MASTERHOST:PORT is the address of the master instance. SLAVENAME and PASSWORD are arbitrary values used to restrict access to the buildslave. After the command is run, the buildslave script will tell you to configure two files, one with a description of the slave, and one with the name and email address of the server admin.

Configuration of the master instance is a bit more complicated. The master configuration includes the configuration of how the builds will be done, as well as configuration of the web interface and the change source configuration. The change source determines how buildbot knows when something has changed. Even with all these options, the master configuration is relatively simple, basically only as complex as you need it to be. For the most part, the defaults work. To get started with the default configuration run the buildbot script with the correct arguments:

```
buildbot create-master -r BASEDIR
```

This will create the default configuration in BASEDIR. The first thing to do is to set up the builders, the code that performs the actions desired by the user, so that they do what they are supposed to. This is accomplished by creating builder factories, the Python objects which generate new builders, which are used to generate builders whenever a build is needed. The factories are specified in terms of steps that are needed to create the builds. Each step is a python class that is part of the buildbot codebase or may be written by the user. Steps are added to the factory in the sequence in which they are to occur. Builders are associated with the factories, and then the schedulers so that when the scheduled event is triggered it knows which builder to use. The web interface is somewhat configurable, but only permissions and whether or not to show buttons to control slave and master instances can be changed. The scheduler allows configuration of what triggers builds, using any of the methods mentioned before as well as custom-made change sources.



Tree is closed (bad bot script checked in)

Builds: continuous | symbols | status

Dashboards: perf | memory | sizes | coverage | flakiness | stats

Chromium: sources | reviews | bugs | dev | support

Sheriffs: aarya, rdsmith, pinkerton, tonyg(WebKit), stuartmorgan(Memory)

Navigate: about | customize | try | experimental | waterfall | failures | console | memory | chromiumos

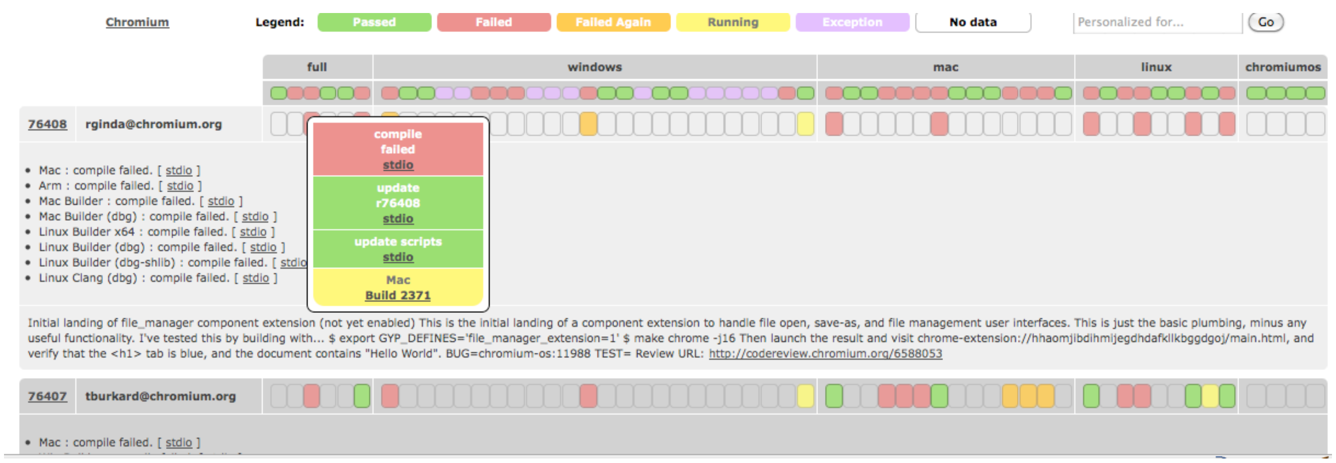
Reliability
76390 : 76390
Crashes:
0 / 40359 URLs
1 / 7240 UI Ops
Last green: 76385

Chromium
Webkit
Perf
Official
Memory

Chromium last build		build successful	failed reliability: partial result of current build	failed compile	build successful	build successful	failed compile	failed compile	build successful	build successful	faile extract_
current activity		building ETA in ~ 31 mins at 12:08 3 pending	idle	idle	building ETA in ~ 13 mins at 11:49 3 pending	building ETA in ~ 15 mins at 11:52 3 pending	idle	idle	building ETA in ~ 3 mins at 11:40 2 pending	building ETA in ~ 8 mins at 11:44 1 pending	offlin
time (PST)	changes	Win	Win Reliability	Mac	Linux	Linux x64	Arm	Win Builder	XP Tests (1)	XP Tests (2)	Vista Test
		compiling stdio			running browser_tests stdio	running browser_tests stdio			running browser_tests stdio	running ui_tests stdio	
11:36:22											
11:36:11											

Figure 11: Buildbot Waterfall View

The buildbot web interface in Figure 11 is the main point of interaction most users have. Buildbot has several views for analyzing the output and status of various builds. The one that Google focuses on is the Waterfall view, which shows all the build configurations and their status', as well as any changes to the source. This view is good for quickly gleaning the overall status of the project. In fact, Google uses the Waterfall view as one of the primary information points as to whether they should close the source tree to changes. Another popular view is the console view, which presents a more change focused version of the status. It displays each commit on a separate line along with the status of the build it triggered, which can be clicked to see the waterfall view in a popup, shown in Figure 12.



76408 rginda@chromium.org

- Mac : compile failed. [stdio]
- Arm : compile failed. [stdio]
- Mac Builder : compile failed. [stdio]
- Mac Builder (dbg) : compile failed. [stdio]
- Linux Builder x64 : compile failed. [stdio]
- Linux Builder (dbg) : compile failed. [stdio]
- Linux Builder (dbg-shlib) : compile failed. [stdio
- Linux Clang (dbg) : compile failed. [stdio]

Initial landing of file_manager component extension (not yet enabled) This is the initial landing of a component extension to handle file open, save-as, and file management user interfaces. This is just the basic plumbing, minus any useful functionality. I've tested this by building with... $ export GYP_DEFINES='file_manager_extension=1' $ make chrome -j16 Then launch the result and visit chrome-extension://hhaomjibdihmijegdhdafkllkbggdgoj/main.html, and verify that the <h1> tab is blue, and the document contains "Hello World". BUG=chromium-os:11988 TEST= Review URL: http://codereview.chromium.org/6588053

76407 tburkard@chromium.org

- Mac : compile failed. [stdio]

Figure 12 - Buildbot Console View

3.3 Improving Build Infrastructure

The time required to build Chromium OS images quickly became an issue for the team. As such, we were asked to build faster build computers, systems designated for compiling the Chromium OS software. While not technically a deliverable for the week, it was necessary for the team's overall success as well as a goal for the project. Two new computers were built and had the development environment installed on them, while a third computer needed RAM before more could be done with it. The fastest of these new computers cut the total build time from about 6 hours to about 3.5 hours, removing a huge bottleneck for the team. In addition, after the completion of our MQP, these computers will serve as build computers for the development team, since some of the current build computers take up to 12 hours to complete one build. Building the computers was largely a process of debugging broken and incompatible parts as well as locating appropriate hardware. The hardware was spread throughout the office from boxes next to the mentors' desks to the server room. In addition, one computer in particular which should have been fast was inexplicably slow despite its capable hardware. Although some time was spent trying to debug the issue there, the results were inconclusive.

3.4 NFS (Network File System) Boot

In order to improve the process of testing new images on our test boards, we were tasked with finding a way to automate the rollout of new filesystem images onto the test boards on a nightly basis. Typically, one would access the local shares and load a nightly build/image onto a USB stick, and then mount this image onto a test board. By automating this process, we would eliminate the need for this manual step.

We chose to use the "NFS root" method, which allows one's target machine (the test board) to use a NFS mount on your host Linux machine as its root filesystem (rootfs). While the NFS root process requires a lengthy setup and causes the target machine to have longer round trip times to the rootfs, it has several advantages as well. Using an NFS share as the rootfs allows changes to be made to the rootfs more easily. Only the filesystem on the NFS share needs to be edited and the changes will be propagated to the target machine. This is a much faster process than loading the changes onto a USB device and rebooting the target machine using that filesystem. In addition, since the filesystem is stored on a remote host, space is rarely, if ever, a concern, since it is simple to add more storage to a server or workstation. This is particularly useful when working with debug images, which are typically much larger than standard images. [17]

Setting up a development board to use an NFS share as its rootfs is non-trivial. The first thing we had to do was set up an NFS server on our host. This was simply done with the commands shown in Figure 13, which enabled the NFS server built into our kernel with several new modules (including nfds, exporfs, lockd, etc.) and then set the base for our NFS exports. Note that the Internet Protocol (IP) address and subnet reflect the IP range that has access to the NFS share. This means that the target machine must be within the 172.17.151.0 to 172.17.151.255 IP range in order to be able to access the NFS share.

```
sudo apt-get install nfs-kernel-server

/export 172.17.151.0/255.255.255.0(rw,fsid=0,no_subtree_check,async)

/export/nfsroot
172.17.151.0/255.255.255.0(rw,nohide,no_subtree_check,async,no_root_squash)
```

Figure 13: Setting up NFS Server

The second export command makes our nfsroot directory (which will contain our root filesystem) accessible as an export, with several options configured. In order to make the rootfs appear in "/export/nfsroot", we used a bind mount to paste the true location with the following set of commands. This is shown in Figure 14.

```
# go to the directory with the latest build
cd chromiumos/chromiumos.git/src/build/images/tegra2_seaboard/latest

# mount it into /tmp/m
chromiumos/chromiumos.git /src/scripts/mount_gpt_image.sh -f . -i
chromiumos_test_image.bin

# copy out the contents of the image
sudo cp -a /tmp/m nfsroot

# unmount the image from /tmp/m
chromiumos/chromiumos.git /src/scripts/mount_gpt_image.sh –u

chromiumos/chromiumos.git/src/build/images/tegra2_seaboard/latest/nfsroot
/export/nfsroot    none    bind    0 0

$ sudo mount /export/nfsroot

$ ls /export/nfsroot/
bin   dev  home  lost+found  mnt  postinst  root  share  tmp     usr
boot  etc  lib   media       opt  proc      sbin  sys    u-boot  var
```

Figure 14: Bind Mounting the Root FS

After restarting the NFS kernel server and disabling the standard firewall in Chromium OS, we were able to successfully set up our NFS server.

Next, we had to make sure we were building a suitable kernel. This required making sure that a suitable network driver was compiled in, and that all the necessary network filesystem options were enabled. The table below shows a few of the important options and what they mean.

- CONFIG_USB_USBNET = enables the USB network subsystem
- CONFIG_NET_AX8817X = enables the particular USB driver
- CONFIG_NETWORK_FILESYSTEMS = enables network filesystem support
- CONFIG_NFS_COMMON, CONFIG_NFS_FS = enables the NFS client in the kernel
- CONFIG_ROOT_NFS = enables the NFS root function

To enable the options, we simply had to edit a configuration file with the following options (Figure 15).

```
chromeos/config/armel/config.flavour.chromeos-tegra2:
CONFIG_USB_NET_AX8817X=y

chromeos/config/config.common.chromeos:
+CONFIG_DNOTIFY=y
+CONFIG_DNS_RESOLVER=y
+CONFIG_LOCKD=y
+CONFIG_LOCKD_V4=y
+CONFIG_NETWORK_FILESYSTEMS=y
+CONFIG_NFSD=m
+CONFIG_NFSD_V3=y
+CONFIG_NFSD_V4=y
+CONFIG_NFS_COMMON=y
+CONFIG_NFS_FS=y
+CONFIG_NFS_USE_KERNEL_DNS=y
+CONFIG_NFS_V3=y
+CONFIG_NFS_V4=y
+CONFIG_ROOT_NFS=y
+CONFIG_RPCSEC_GSS_KRB5=y
+CONFIG_SUNRPC=y
+CONFIG_SUNRPC_GSS=y
+CONFIG_USB_USBNET=y
```

Figure 15: Kernel Configuration

The final step in getting an NFS boot to work on a target machine was to create a boot command with the necessary environment variables. On ARM systems, like the systems we are working with, U-passes the kernel parameters to the kernel. Normally, U-Boot would use NAND or eMMC (Embedded Multimedia Card – flash memory storage located in the seaboard) as the root filesystem however, for our task that needed to be changed. We had to change the root option and add two new options, nfsroot and ip. The boot command, as entered on the seaboard, loads the kernel from the USB and then boots it with the rootfs that is provided through the NFS server, our boot command is shown in Figure 16 below.

```
Tegra2 (SeaBoard) # setenv myargs=setenv bootargs ${mem} video=${videospec}
console=${console} usbcore.old_scheme_first=1 tegraboot=${tegraboot} ${lp0_vec}
tegrap earlyprintk root=/dev/nfs4 nfsroot=172.17.151.39:/exoirt.nfsroot ip=dhcp
rw rootwait

Tegra2 (SeaBoard) # setenv myboot=usb start \; ext2load usb 0:3 ${loadaddr}
/boot/${bootfile} \; run myargs; bootm ${loadaddr}

Tegra2 (SeaBoard) # print myboot
myboot=usb start; ext2load usb 0:3 ${loadaddr} /boot/${bootfile}; run myargs;
bootm ${loadaddr}
```

Figure 16: NFS Boot Commands

Enabling NFS boot to work with our host and target machines, as shown below with a partial boot trace, takes away the need to manually load nightly images onto our test boards, and allows a developer to make use of the large amount of disc space available on the host machines.

```
## Booting kernel from Legacy Image at 00408000 ...
   Image Name:   Linux-2.6.36
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3514264 Bytes = 3.4 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 2.6.36 (sjg@kiwi.mtv.corp.google.com) (gcc version
4.4.3 (Gentoo Hardened 4.4.3-r4 p1.2, pie-0.4.1) ) #27 SMP PREEMPT Thu Dec 23
14:05:05 PST 2010
[    0.000000] CPU: ARMv7 Processor [411fc090] revision 0 (ARMv7), cr=10c53c7f
[    0.000000] CPU: VIPT nonaliasing data cache, VIPT nonaliasing instruction
cache
[    0.000000] Machine: seaboard
[    0.000000] Ignoring unrecognised tag 0x54410008
[    0.000000] Memory policy: ECC disabled, Data cache writealloc
[    0.000000] PERCPU: Embedded 8 pages/cpu @c0fcd000 s9312 r8192 d15264 u65536
[    0.000000] pcpu-alloc: s9312 r8192 d15264 u65536 alloc=16*4096
[    0.000000] pcpu-alloc: [0] 0 [0] 1
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total
pages: 227328
[    0.000000] Kernel command line: mem=384M@0M nvmem=128M@384M mem=512M@512M
video=tegrafb console=ttyS0,115200n8 tegraboot=nand
tegrapart=system:3680:2bc0:800 smp root=/dev/nfs4
nfsroot=172.17.151.39:/export/nfsroot ip=dhcp rw rootwait
[    0.000000] PID hash table entries: 4096 (order: 2, 16384 bytes)
[    0.000000] Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
[    0.000000] Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)

...
```

```
[   20.793312] usb 1-1.1: New USB device found, idVendor=0b95, idProduct=7720
[   20.800425] usb 1-1.1: New USB device strings: Mfr=1, Product=2,
SerialNumber=3
[   20.807928] usb 1-1.1: Product: AX88x72A
[   20.812182] regulator_init_complete: incomplete constraints, leaving REG-
SM_0 on
[   20.819913] usb 1-1.1: Manufacturer: ASIX Elec. Corp.
[   20.825096] usb 1-1.1: SerialNumber: 000001
[   21.715416] asix 1-1.1:1.0: eth0: register 'asix' at usb-tegra-ehci.2-1.1,
ASIX AX88772 USB 2.0 Ethernet, 68:7f:74:9f:f6:f2

...

[   22.377062] ADDRCONF(NETDEV_UP): eth0: link is not ready

...

[   23.513408] ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[   23.517802] eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
[   25.420077] ., OK
[   25.450201] IP-Config: Got DHCP answer from 172.16.2.3, my address is
172.16.2.2
[   25.462474] IP-Config: Complete:
[   25.465579]      device=eth0, addr=172.16.2.2, mask=255.255.255.0,
gw=172.16.2.1,
[   25.473686]      host=seaboard0, domain=mydomain.com, nis-domain=(none),
[   25.482878]      bootserver=172.16.2.3 rootserver=172.16.2.3, rootpath=
[   25.492066] Looking up port of RPC 100003/2 on 172.16.2.3
[   25.532272] Looking up port of RPC 100005/1 on 172.16.2.3
[   25.565966] VFS: Mounted root (nfs filesystem) on device 0:16.
[   25.605134] devtmpfs: mounted
[   25.608453] Freeing init memory: 216K
[   25.629564] Not activating Mandatory Access Control now since /sbin/tomoyo-
init doesn't exist.
Developer Console

To return to the browser, press:

  [ Ctrl ] and [ Alt ] a[   30.996904] unknown ioctl code
nd [ F1 ]

To use this console[   31.000639] NvRmIoctls_NvRmFbControl: deprecated
, the developer mode switch must be engaged.
Doing so will destroy any saved data on the system.

In developer mode, it is possible to
- login and sudo as user 'chronos'
- require a password for sudo and login(*)
- disable power management behavior (screen dimming):
  sudo initctl stop powerd
- install your own operating system image!

* To set a password for 'chronos', run the following as root:

echo "chronos:$(openssl passwd -1)" >
/mnt/stateful_partition/etc/devmode.passwd

Have fun and send patches!

localhost login:
```

Figure 17: NFS Boot Console Output

A complete HowTo for NFS Boot can be seen in Appendices II and III.

3.5 DHCP (Dynamic Host Configuration Protocol) Boot

DHCP is an auto configuration protocol that is used on IP (Internet Protocol) networks. Any computer that is connected to an IP network must be set up with an "address" before it can communicate with other computers on the network. DHCP allows a computer to be set up automatically while keeping track of all of the computers actively using the network and preventing two computers from accidentally using the same IP address. DHCP Boot, which uses a DHCP server to automatically assign an IP address to the target board, is very similar to NFS Boot because it uses a mounted root filesystem that is not pre-loaded onto the test board. It further improves on NFS Boot as a means of automation, though, because it loads the system kernel over a TFTP (Trivial File Transfer Protocol) server rather than from a USB device attached to the test board as with NFS Boot. TFTP is a file transfer method that is commonly used for the transfer of boot or configuration files over a network. TFTP can be implemented without using much memory, and is still reliable. For this reason, TFTP was our choice for transferring the kernel over the network. Using DHCP Boot with a TFTP server allows a user to load both the kernel and root filesystem over the network. [18]

The configuration for DHCP Boot does not require many more steps than NFS Boot. First a user must set up a DHCP Server and a TFTP Server. The DHCP Server is used to provide IP addresses to targets on the network, and the TFTP Server is used to send the kernel to U-Boot (the bootloader) when it requests one. After setting up these servers, a user needs to define some boot arguments such as the server IP addresses, specific bootfiles, and a path to the kernel on the host machine. With this setup, a user can boot their test board with the kernel and root filesystem completely loaded over a local network.

A complete HowTo for DHCP Boot can be seen in Appendix IV.

3.6 Autotest

NVIDIA currently has a large suite of proprietary tests, called the NVtest suite, that are run on their development boards to ensure the software is running properly. These proprietary tests are currently run one-by-one by testers in NVIDIA's Quality Assurance group. Our group was tasked with automating these tests. Chromium OS uses a test harness called Autotest to automatically run its test suite. Autotest allows tests to be run on remote devices in a repeatable, parallelizable, and efficient way [14]. Automating these tests frees up Quality Assurance resources for better and more in-depth testing. In addition, automating these tests allows developers to get more rapid feedback to their changes. Faster feedback means that issues can be found before the developer merges their changes with the main tree, preventing their bugs from propagating to the rest of the developers.

There were two parts to this deliverable: figuring out how to run NVIDIA's proprietary tests, and writing the adaptor that allows these tests to be run by Autotest. Once we figured out how to run a few tests, it became clear that automating creation of the adaptors would benefit the team greatly. In order to accomplish our deliverable of having a working NVtest running through Autotest, the team created a program that generates the adaptor. The generator was created successfully and allowed the team to rapidly port NVIDIA's proprietary tests to Autotest.

To learn how to run the tests, the group looked through documented bug reports from the NVidia Quality Assurance team to see if information on running each test was available. The group then looked at the source code of each test to see what was required to run it in the case that the test required parameters to be included. To the disappointment of the team, only sixty five percent of the tests were able to be catalogued based on the aforementioned methods, so a letter was drafted and sent to the NVidia Quality Assurance team to ask them how to run the remaining NVtests on the Seaboard.

Autotest allows tests to be run on computers connected over the network. In this regard, Autotest has a client-server architecture. There is a target board, which is the hardware upon which the tests are to be run, and a test server, typically a developer's computer which sends the tests over the network to the target board. We realized that this architecture required that the target board be online to run the NVtests through Autotest, so that the test server could access the target board. After this was done the team used the scripts NVidia has written for using Autotest to run the Autotests we had created. This works correctly and now testing can be done from several computers onto one board simultaneously.

Autotests consist of two different files; a control file [Figure 18], and a command file[Figure 19][14]. The control file specifies which command files will be run as well as any other execution commands that are required to execute the command files, such as commands to reboot the target machine. Information required in the control file includes the author, documentation, test name, time, test class, test category, and test type. Author should include contact information for the writer of the command files and control file. Doc is a description of the test, including the arguments that can be passed to it. The name variable is just the name of the test, while the time variable is how long the test should take to run. Short is less than fifteen minutes, medium is less than four hours, and long is more than four hours. Test_class describes where the test belongs. In our case we made it general, but more specific cases might be kernel, or even hardware tests. Test_category tells if the test is for stress, functionality or otherwise. Test_type indicates if it is a client, server, or multi-client test. The last line of Figure 18 shows a command to execute a job. There are two types of jobs; simple jobs, and phased jobs. Simple jobs are sets of straightforward commands directed at driving execution of tests, while phased jobs are used in cases such as where the system needs to be rebooted in between each test case.

```
AUTHOR = "name  name@yahoo.com"
TIME = "SHORT"
NAME = 'HelloWorld'
DOC = """
Runs the HelloWorld test
"""
TEST_TYPE = 'CLIENT'
TEST_CLASS = 'General'
TEST_CATEGORY = 'Functional'
```

```
Job.run_test('HelloWorld')
```

Figure 18: Autotest Control File

The actual command files contain commands that are sent to and run on the target machine. There are several types of tests and among those we used functional and performance Autotest tests. When Autotest runs a functional test the test files parse the output to see if the test either passed or failed. In the case of performance tests, the output can be parsed for information such as performance data. Figure 19 shows the code for automating a hello world test. There are four built in functions when running: initialize, setup, run_once, and postprocess_iteration. Initialize is run every time the test is executed. Setup is only utilized on the first run of the test, and if compilation is needed, this is where it would be called. Run_once, despite what the name implies, is actually run for however many iterations that are specified, but carries the bulk of commands that will be used to test the client machine. Postprocess_iteration handles much of the output back to the server on how the test did. In our case, due to the simplicities of the tests we were automating, we only needed the use of the run_once function. Figure 2 shows an example of an Autotest test, and the format we would use. The command subprocess.Popen() executes the command on the target machine. In the case of NVtest suite, the line 'echo', 'HelloWorld!' would be replaced with the command and the command arguments being passed to it.

```
from autotest_lib.client.bin import test
import subprocess
from subprocess import CalledProcessError
from autotest_lib.client.common_lib import error

class HelloWorld(test.test):

        version = 1

        def check_for_failure(self, output):
                if output.find('ALL TESTS PASSED') == -1:
                        print "Found error"
                        raise error.TestFail('Test Failed')

        def run_once(self):

                proc = subprocess.Popen(['echo', 'HelloWorld!'],
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
                out, err = proc.communicate()
                ret = proc.poll()

                self.check_for_failure(err)

                return ret
```

Figure 19: Autotest Test

3.7 Selenium

The performance tests that the Quality Assurance team had been running were located on several webpages that were run individually by hand. This results in a problem that too much time is being used to run and wait for performance test metrics to be returned. Part of our project was to help automate this process. To automate this process we would need a tool that allowed us to open the browser, navigate to a webpage, depending on the webpage click a button, wait for the performance test to finish running, and then grab the performance metrics. The first program we found was a tool called Selenium. Selenium is a command line driven application that uses several tools to run browser tests remotely [15]. There are two parts to the test; a server which acts as an HTTP proxy for web requests, and the client library which sends commands to the server. The library specifies information such as which browser the tests will be run on, and what will be tested. Appropriate browsers include Internet Explorer, Firefox, Safari, and Chrome. Figure 20 shows the code for a test that starts up Google Chrome and travels to http://www.google.com [Figure 20; Note 1]. It then types in “hello world!” [Figure 20; Note 2] and when the search is executed asserts that the title of the webpage is “hello world! – Google Search” [Figure 20; Note 3]. If a test fails, the name of the test along with why the test failed is printed out. If the test passes than only the test name and the word ‘Ok’ are printed.

```
from selenium import selenium
import unittest, time

class TestHelloWorld(unittest.TestCase):
	def setup(self):
		self.selenium = selenium(“localhost”, \
			4444, “*googlechrome”, “http://www.google.com”)
		self.selenium.start()

	def test_helloworld(self):
		sel = self.selenium
		sel.open(“http://www.google.com”) # [Note 1]
		time.sleep(1)

		sel.type(“q”, “hello world!”) #     [Note 2]
		sel.submit(“f”)

		for i in range(30):
			try:
				if sel.is_element_present(“resultStats”):
					break
				else:
					time.sleep(1)
			except:
				self.fail(“Except: Could not search for element”)
		else:
			self.fail(“Timeout: 30 seconds”)

		self.assertEqual(“hello world! – Google Search”, sel.get_title())
		                                  #  [Note 3]

	def teardown(self):
		self.selenium.stop()

if __name__ == “__main__”
```

```
unittest.main()
```

Figure 20: Selenium Test

Selenium requires Java to run the Selenium-Server, and since Java had not yet been ported over for ARM architecture we could not actually use Selenium in our final implementation of performance testing. We started looking for other web testing frameworks to help automate performance tests. Three solutions we came across were to port Java over for use on ARM, use Selenium-Webdriver, or use a program called PyAuto. Since porting Java over was somewhat impossible to fit into the scope of the project, we decided to explore the other two options. Selenium-Webdriver is a program that would effectively do the same thing that selenium did and the reason we looked into it was because it promised similar functionality to selenium while changing the underlying framework. We initially thought that changing the framework might have meant using something other than Java but to our dismay we realized that it too needed the use of Java on the target platform. Our final solution was to use PyAuto, a python driven program that would allow us to do much that selenium could do.

3.8 PyAuto

PyAuto is a python interface to Chromium's automation framework [16]. It can be used, much like Selenium, to test browser functionally and performance. We decided to use PyAuto for out testing framework because of the obvious benefits it gives. First, since PyAuto uses python, we wouldn't need to port over any major languages. Second, the chromium team had also already done extensive work in getting PyAuto to work with Chromium, meaning that all we would need to do is change a few of the build scripts and then rebuild for our platform. Initially we had problems figuring out what we needed to change in the build script, but after extensive trial an error we got PyAuto to build into Chromium. Those changes were eventually sent upstream to Google allow PyAuto to be built not only for x86 architecture but for ARM architecture as well. Finally, since we used python tests in Selenium, it would be easy to port over the selenium tests we previously wrote.

Below, in Figure 21, we show an example test which first navigates to http://www.google.com [Figure 21; Note 1] and then asserts if the title of the page equals "Google"[Figure 21: Note 2]. This is of course a very simplified case of what PyAuto is capable of. When we implemented our tests we actually needed to run JavaScript on the page, or grab text to send to Autotest; For this we used the command self.GetDOMValue('command'). The 'command' would be in JavaScript, for example, to get data from a performance test that had already run and was included in an element that was identifiable (having id= 'something'), the command 'document.getElementById('identifier').textContent would be acceptable for retrieving that information.

```
import unittest
import pyauto
```

```
class MyTest(pyauto.PyUITest):
  def testNavigation(self):
    self.NavigateToURL("http://www.google.com")          #[Note 1]
    self.assertEqual("Google", self.GetActiveTabTitle()) #[Note 2]


if __name__ == '__main__':
  pyauto.Main()
```

Figure 21: Pyauto Test [16]

One of the only issues we had with PyAuto is the way in which it traverses and returns elements from the DOM (Document Object Model). We believe that PyAuto is finicky using JavaScript when there are multiple elements returned from a query. For example, when there is no identifier in an element holding data that is meant to be retrieved we would need to call self.GetDOMValue("document.getElementsByTagName('DIV').item(3).textContent"). This command is meant to look at the document and get all the elements with DIV tags. From that array it grabs the third element and the returns the containing text of that element. To our dismay, PyAuto would only return a blank string even when the command would be applicable. We hope that as Chromium automation improves, later versions of PyAuto will work consistently.

4.0 Results

Our project was very successful. We were able to successfully build and set up four new build computers for NVIDIA. One of these computers completed a build in about two hours, which compares favorably to the existing build machines, some of which take up to twelve hours. This will allow NVIDIA to perform more builds as well as get faster feedback. This allows them to improve the testing coverage as well as more quickly identify regressions in a commit. These build machines are currently deployed in the team's buildbot farm.

One of the other key goals was to enable NVIDIA to replicate Google's builds. We successfully created a script that is capable of downloading the repo manifest file from Google's buildbot instance, and deployed that script within a buildbot environment. We documented this process for NVIDIA in their internal wiki. They will be deploying the script and new buildbot configuration after we finish our project.

At the offset of our project, PyAuto was not building for the ARM architecture, which the Tegra chips all run. This was due to Chromium OS being released initially on the x86 architecture; so many times things are not as well tested on the ARM architecture as they are on x86. We fixed this and sent the patch for the fix upstream to Google, to be included in future development.

In order to automate running the nvtest suite, we needed to wrap each test in autotest-compatible Python code. Since this code is extremely repetitive, we created a script to generate the files based on input from a text file that contained the command line arguments to run each test. This script was given to our mentor and the tests generated have entered a review process in order to be upstreamed to Google so that it can be made available to Google and NVIDIA's other partners.

Another goal of our project was automating the boot process. NFS Boot and DHCP Boot were the methods tested, and directions on how to run these processes were posted on NVIDIA's internal wiki at the conclusion of our project. By following these steps, an NVIDIA employee can fully boot a test machine over the network, without having the kernel or root filesystem pre-loaded onto the test machine.

One of our final deliverables was to catalog and understand how to run all of the tests in the Nvtest suite. Nvtest suite is a set of proprietary tests that test system functionality. We completed this objective and had documented 100% of the tests about if they could run and if so, how to run them. Unfortunately this information will probably not be used because the Quality Assurance team has a website that displays tests and how to run them. We were also asked to write Autotest wrappers to run these tests from the host machine. We also completed this goal as well as put it into an ebuild and sent it upstream to Google. We are not exactly sure if these wrappers will be used, but we are under the impression that the wrappers will at least provide a way for future testing groups to be able to use information as a resource to write their own.

We were also asked to come up with a performance test tar-ball including some PyAuto tests that navigate to websites and retrieve performance metrics as well as the Autotest wrappers to run these tests. We completed this goal and created a tar-ball which would be used to make an ebuild and sent upstream. Again, we are not sure if these tests will actually be used, but we believe that they may be used as a resource in future cases where automation is required to physically go to a website and grab information from it.

Overall, we completed each goal that was initially set out for us, as well as more. The work we did has helped NVIDIA to automate their building, booting, and testing processes, and has opened the door to further automation of these systems. We, as well as our sponsors at NVIDIA, believe this project was a great success.

5.0 Conclusion

The main goal of our project was to automate the test infrastructure across several builds and revisions of those builds while being able to completely boot an entire operating system over the network. Although we faced many problems throughout the project, we overcame all of these adversities in one way or another and completed all the goals we were originally given as well as the goals we were given at other points in the project. As such, the project should be considered a success. We hope to see our project work help the Quality Assurance team at NVIDIA in their workload as well as the speed at which they can construct helpful reports on the quality of their product. Hopefully, the test infrastructure that we helped create will be built upon and added to in future years as better and faster technologies are deployed.

Works Cited

[1] Dang, Alan. "History of NVIDIA." *FiringSquad: Home of the Hardcore Gamer - Games, Hardware, Reviews and News*. 9 Feb. 2001. Web. 18 Nov. 2010. <http://www.firingsquad.com/features/nvidiahistory/>.

[2] Anonymous "NVIDIA Corporation -- Company History." *Find Funding with Banks, Investors, and Other Funding Sources | FundingUniverse*. Web. 18 Nov. 2010. <http://www.fundinguniverse.com/company-histories/NVIDIA-Corporation-Company-History.html>.

[3] "Chromium OS." *The Chromium Projects*. Web. 18 Nov. 2010. <http://www.chromium.org/chromium-os>.

[4] Anonymous "Mobile." *Welcome to NVIDIA - World Leader in Visual Computing Technologies*. Web. 30 Nov. 2010. <http://www.nvidia.com/object/tegra.html>.

[5] Anonymous. "Making Sense of Smartphone Processors: The Mobile CPU/GPU Guide." Web log post. *TechAutos*. Web. 06 Dec. 2010. <http://www.techautos.com/2010/03/14/smartphone-processor-guide/>.

[6] Anonymous. "What's New in Performance and Reliability Monitoring in Windows Server 2008 R2 and Windows 7." *Microsoft TechNet: Resources for IT Professionals*. Web. 06 Dec. 2010. <http://technet.microsoft.com/en-us/library/ee731897(WS.10).aspx>.

[7] Turnbull, Giles. "What Is Activity Monitor (or How to Take Your Mac's Pulse) - O'Reilly Media." *MacDevCenter.com-- Macintosh Development, Open Source Development*. Web. 02 Dec. 2010. <http://macdevcenter.com/pub/a/mac/2005/10/04/activity-monitor.html>.

[8] Anonymous "MPG - Mac Performance 101: Cores, Processes, Memory - Observing System Performance with Activity Monitor." *Macintosh Performance Guide Blog*. Web. 02 Dec. 2010. <http://macperformanceguide.com/Mac-MonitoringTips.html>.

[9] Brian K. Tanaka, "Monitoring Virtual Memory with vmstat." *Linux Journal*, Web. 02 Dec. 2010. <http://www.linuxjournal.com/article/8178>.

[10] Henry Ware <al172@yfn.ysu.edu>, Fabian Frédérick ffrederick@users.sourceforge.net, "vmstat." *vmstat manual page*, Web. 02 Dec. 2010.

[11] Andre Lewis, "Understanding Linux CPU Load – when should you be worried?" *Scout*, Web. 06 Dec. 2010. http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages

[12] Jim/James C. Warner <warnerjc@worldnet.att.net>, “top.” *top manual page*, Web. 02 Dec. 2010.

[13] “Buildbot Manual – 0.8.3.” *Buildbot*. Web. 15 Mar. 2011. <http://buildbot.net/buildbot/docs/current/>.

[14] Software., Edgewall. *Autotest – Trac*. Web. 28 Jan. 2011. <http://autotest.kernel.org/wiki>.

[15] "Selenium Remote-Control." *Selenium Web Application Testing System*. Web. 28 Jan. 2011. <http://seleniumhq.org/projects/remote-control/>.

[16] "PyAuto: Python Interface to Automation." *The Chromium Projects*. Web. 27 Feb. 2011. <http://www.chromium.org/developers/testing/pyauto>.

[17] "NFS-Root Mini-HOWTO." *Internet FAQ Archives - Online Education - Faqs.org*. Web. 01 Mar. 2011. <http://www.faqs.org/docs/Linux-mini/NFS-Root.html>.

[18] "DHCP FAQ." *The DHCP Handbook*. Web. 01 Mar. 2011. <http://www.dhcp-handbook.com/dhcp_faq.html>.

Appendices

Appendix I – Project Schedule

Week of 1/3

- Get the machines setup – ***Completed***
- Do a ChromiumOS build. - ***Completed***
- Be able to load that build onto the seaboard. – ***Completed***
- Loading a nightly image (both normal and autotest images). – ***Completed***

Week of 1/10 (Deliverable 1 on Friday)

- NFS Boot to eMMC – ***Completed***
- Run an autotest test – ***Completed***
- Improve performance of build infrastructure – ***Completed***
- Catalogue and understand each of the NV tests - ***Completed***
- Get autotest with some type of dummy autotest or 'hello world' autotest test running – ***Completed***
- Get one of the NV specific test run through autotest. - ***Completed***
- Get a list of performance tests we care about from the chromiumos team and start cataloging them. – ***Completed***

Week 1/17

- Convert nvtest suite to autotest - ***Completed***
- Begin performance testing using autotest - ***Completed***
- Continue bringing up new build systems - ***Completed***

Week 1/24

- Grok buildbot system - ***Completed***
- Work on autotest performance testing - ***Completed***
- Continue bringing up new build systems - ***Completed***

Week 1/31 (Deliverable 2 on Friday)

- Complete autotest performance tests - ***Completed***
- Finish bringing up new build systems - ***Completed***
- Work on build bot system - ***Completed***

Week 2/7

- Work on build bot system – ***Completed***
- Get PyAuto browser tests working - ***Completed***

Week 2/14

- Have a working buildbot repo manifest file - ***Completed***
- Automate repo manifest file with nVidia buildbot - ***Completed***

Week 2/21 (Deliverable 3 on Friday) ** LAST DELIVERABLE **

- Investigate and implement the possibility of using distcc for builds - ***Completed***
- Have all deliverables ready to demonstrate – ***Completed***

- Get DHCP boot working - ***Completed***

Week 2/28

- Write project report - ***Completed***
- Finish up loose ends - ***Completed***

Appendix II – NFS Boot - HowTo

Modified from Google instructions at: http://www.chromium.org/chromium-os/how-tos-and-troubleshooting/debugging-tips/host-file-access

1) Start with a chromium tree with a working ChromiumOS build.

2) Verify that this build image boots correctly on your test board before making the following changes, else re-build.

3) Set up an NFS server on your Linux host machine

First, install the NFS server package (these instructions are for Ubuntu 10.04 Lucid). This enables the NFS server built into your kernel. When it starts you may notice that a number of new modules have been loaded into your kernel (nfsd, exportfs, lockd, etc.)

```
  $ sudo apt-get install nfs-kernel-server
```

4) The server needs to know which directories you want to 'export' for clients. This is specified in the /etc/exports file. Modify to look somewhat like this:

```
  $ /export             172.16.0.0/16(rw,fsid=0,no_subtree_check,async)
  $ /export/nfsroot
172.16.0.0/16(rw,nohide,no_subtree_check,async,no_root_squash)
```

The first entry sets the base of the NFS exports. The second entry is the nfsroot directory which will contain your root filesystem. This is the directory that the client will see when it mounts the NFS root. The IP address should be changed to match your local setup.

A number of options are provided, briefly:

- rw - the target will have both read and write access. You can also use ro for readonly but the system will not boot with a read-only root filesystem (without a bit of work!)
- fsid=0 - tells NFS that this is the root of all exported filesystems
- no_subtree_check disables checking for accesses outside the exported portion of a filesystem. This speeds up and simplifies things for the client and server.
- async - requests are acknowledged before data is actually written. For example if the client writes to a file, the server will respond that the write has completed, and then continue in the background to actually do the write, perhaps to a disc drive. This improves performance.
- no_root_squash - the target can access files as root, with full unrestricted permissions. This is important for the root filesystem because the kernel would otherwise not have access to devices in /dev, log files in /var/log, etc.

- nohide - tells the NFS server to show the contents of a directory even if it is mounted from elsewhere

5) Next we need to make the root filesystem appear in /export/nfsroot. The following steps guide us through bind mounting the true location onto /export/nfsroot. First we need to unpack a suitable image.

Let's assume that you have your Chromium trunk directory as ~/chromiumos/chromiumos.git and you are using a tegra2_seaboard build:

```
  # go to the directory with the latest build
  $ cd ${TREE_ROOT}/src/build/images/tegra2_seaboard/latest
  # mount it into /tmp/m
  $ ${TREE_ROOT}/src/scripts/mount_gpt_image.sh -f . -i chromiumos_image.bin
  # copy out the contents of the image
  $ sudo cp -a /tmp/m nfsroot
  # unmount the image from /tmp/m
  $ ${TREE_ROOT}/src/scripts/mount_gpt_image.sh -u
```

This will put a full copy of the build image root disc into ~/chromiumos/chromiumos.git/src/build/images/tegra2_seaboard/latest/nfsroot

6) Now we need to make it appear in /export/nfsroot. Edit your /etc/fstab file with the full path:

```
  $ $TREE_ROOT/src/build/images/tegra2_seaboard/latest/nfsroot
/export/nfsroot    none    bind    0 0
```

Note this must appear all on one line and you can use tabs or spaces between fields. This will be activated automatically when your server reboots, but since it is already running, ask it to mount this now. After the mount you will see that the root filesystem has appeared at /export/nfsroot as desired.

```
  $ sudo mount $TREE_ROOT/src/build/images/tegra2_seaboard/latest/nfsroot
  $ ls /export/nfsroot/
  bin   dev  home  lost+found  mnt  postinst  root  share  tmp     usr
  boot  etc  lib   media       opt  proc      sbin  sys    u-boot  var
```

7) Verify that /etc/idmapd.conf is correct, as follows.

```
  [General]

  Verbosity = 0
  Pipefs-Directory = /var/lib/nfs/rpc_pipefs
  Domain = local.domain.edu

  [Mapping]

  Nobody-User = nobody
  Nobody-Group = nogroup
```

Restart the NFS kernel server.

```
  $ sudo /etc/init.d/nfs-kernel-server restart
   * Stopping NFS kernel daemon                                          [
OK ]
   * Unexporting directories for NFS kernel daemon...                    [
OK ]
   * Exporting directories for NFS kernel daemon...                      [
OK ]
   * Starting NFS kernel daemon                                          [
OK ]
```

8) Finally, the standard firewall setup in Chromium OS does not permit NFS. You may wish to simply disable the firewall - the /export/nfsroot/etc/init/iptables.conf file should be changed to do this. At the top, change the start on line to start on never. If you don't do this you will likely boot to a login prompt, but then you will see 'NFS server not responding' messages once the firewall kicks in.

9) Build a suitable kernel

There are quite a few options that you need to enable in the kernel to support NFS root. First you need to make sure that a suitable network driver is compiled in, and secondly you need to enable all the network filesystem options. It is very important to verify that your network driver is set with the “=y” option in your configuration files, otherwise you will not be able to mount the rootfs when you try to boot. The following list for a USB network adapter setup gives you an idea of what is required. Some of the important options are:

- CONFIG_USB_USBNET - enables the USB network subsystem
- CONFIG_NETWORK_FILESYSTEMS - enables network filesystem support
- CONFIG_NFS_COMMON, CONFIG_NFS_FS - enable NFS client in kernel
- CONFIG_ROOT_NFS - enable NFS root function

Note: some of the options below are required for NFS root, some for NFS mounting and some for NFS serving. Enter the chroot and run:

```
  emerge-tegra2_seaboard kernel-next
```

Then edit the config files as follows:

```
  chromeos/config/armel/config.flavour.chromeos-tegra2:
  CONFIG_USB_NET_AX8817X=y
  chromeos/config/config.common.chromeos:
  +CONFIG_DNOTIFY=y
  +CONFIG_DNS_RESOLVER=y
  +CONFIG_LOCKD=y
  +CONFIG_LOCKD_V4=y
  +CONFIG_NETWORK_FILESYSTEMS=y
  +CONFIG_NFSD=m
  +CONFIG_NFSD_V3=y
  +CONFIG_NFSD_V4=y
  +CONFIG_NFS_COMMON=y
  +CONFIG_NFS_FS=y
  +CONFIG_NFS_USE_KERNEL_DNS=y
  +CONFIG_NFS_V3=y
  +CONFIG_NFS_V4=y
  +CONFIG_ROOT_NFS=y
  +CONFIG_RPCSEC_GSS_KRB5=y
  +CONFIG_SUNRPC=y
  +CONFIG_SUNRPC_GSS=y
  +CONFIG_USB_USBNET=y
```

Run the following sequence of unmerging/merging

```
  emerge-tegra2_seaboard kernel-next --unmerge
  emerge-tegra2_seaboard kernel-next
```

10) Re-build your image with the modifications (make_image step).

11) Load that build onto the USB device and follow the steps at https://wiki.nvidia.com/wmpwiki/index.php/WMP_ap20/ChromeOS/Build#Manually_Updating_the_Kernel so that the kernel on the 3rd partition is used.

12) (OPTIONAL): Clean the 3rd partition out except for the /boot folder to ensure that you really are booting off of NFS.

13) Set up the Boot Loader

On ARM systems, U-Boot is responsible for passing the kernel parameters to the kernel. Normally, U-Boot will use NAND or eMMC as the root filesystem, so this needs to be changed. Since Chrome OS does not presently make use of an environment save area, the easiest approach is to rebuild U-Boot with this new environment. But first you can test it. Start U-Boot with a suitable image on a USB stick and press a key to break in before it boots Linux. Then look at a few environment variables (these may differ depending on your board/configuration):

It is starting up USB, loading a kernel into memory, then setting the bootargs (kernel parameters), then booting the kernel with bootm.

You can create your own boot command with something like:

```
  Tegra2 (SeaBoard) # setenv myargs setenv bootargs mem=384M@0M
nvmem=128M@384M mem=512M@512M video=tegrafb console=ttyS0,115200n8
usbcore.old_scheme_first=1 tegraboot=nand ${lp0_vec} tegrap earlyprintk
root=/dev/nfs nfsroot=<host-ip-address>:/export/nfsroot ip=dhcp rw rootwait
  Tegra2 (SeaBoard) # setenv myboot usb start \; ext2load usb 0:3 ${loadaddr}
/boot/vmlinux.uimg \; run myargs \; bootm ${loadaddr}
```

Note that we have changed the root option and added two new options (nfsroot and ip). The IP address of your server needs to go after nfsroot= and you will need a DHCP server running also. Note: it is possible for NFS to get the server details from your DHCP server (exercise for reader); going the other way, it is also possible to specify a fixed IP address and not need a DHCP server.

The “bootfile” for tegra2_seaboard is vmlinux.uimg

Now try booting it:

```
  Tegra2 (SeaBoard) # run myboot
```

With a bit of luck it will load the kernel from USB, then boot it with your NFS server providing the root.

In order to make the above automatic, you can edit the default scripts in U-Boot to add your changes. For example, for SeaBoard the appropriate file is include/config/tegra2_common.h. Then rebuild U-Boot and update it on your board.

A partial boot trace is shown below to show the sequence of events:

```
  CrOS> setenv myargs setenv bootargs mem=384M@0M nvmem=128M@384M
mem=512M@512M video=tegrafb console=ttyS0,115200n8
  usbcore.old_scheme_first=1 tegraboot=nand ${lp0_vec} tegrap earlyprintk
root=/dev/nfs nfsroot=172.17.149.151:/export/nfsroot
  ip=dhcp rw rootwait
  CrOS> setenv myboot usb start \; ext2load usb 0:3 ${loadaddr}
/boot/vmlinux.uimg \; run myargs \; bootm ${loadaddr}
  CrOS> run myboot
  USB:   Tegra ehci init hccr c5008100 and hcor c5008140 hc_length 64
  Register 10011 NbrPorts 1
  USB EHCI 1.00
  scanning bus for devices... 6 USB Device(s) found
  scanning bus for storage devices... 1 Storage Device(s) found
  Loading file "/boot/vmlinux.uimg" from usb device 0:3 (gpt3)
  3568840 bytes read
  ## Booting kernel from Legacy Image at 0040c000 ...
  Image Name:   kernel
  Image Type:   ARM Linux Kernel Image (uncompressed)
  Data Size:    3568776 Bytes = 3.4 MiB
  Load Address: 10008000
  Entry Point:  10008000
  Verifying Checksum ... OK
  Loading Kernel Image ... OK
  OK
  Starting kernel ...
  Uncompressing Linux... done, booting the kernel.
  [    0.000000] Initializing cgroup subsys cpuset
  [    0.000000] Initializing cgroup subsys cpu
  [    0.000000] Linux version 2.6.37 (snagpal@snagpal) (gcc version 4.4.3
(Gentoo Hardened 4.4.3-r4 p1.2, pie-0.4.1) ) #1 SMP
  PREEMPT Fri Jan 28 15:41:16 PST 2011
  [    0.000000] CPU: ARMv7 Processor [411fc090] revision 0 (ARMv7),
cr=10c53c7f
```

```
  [    0.000000] CPU: VIPT nonaliasing data cache, VIPT aliasing instruction
cache
  [    0.000000] Machine: seaboard
  [    0.000000] Ignoring unrecognised tag 0x54410008
  [    0.000000] bootconsole [earlycon0] enabled
  [    0.000000] Memory policy: ECC disabled, Data cache writealloc
  [    0.000000] PERCPU: Embedded 7 pages/cpu @c0f94000 s7264 r8192 d13216
u32768
  [    0.000000] Built 1 zonelists in Zone order, mobility grouping on.
Total pages: 227328
  [    0.000000] Kernel command line: mem=384M nvmem=128M@384M mem=512M@512M
video=tegrafb console=ttyS0,115200n8
  usbcore.old_scheme_first=1 tegraboot=nand tegrap earlyprintk root=/dev/nfs
nfsroot=172.17.149.151:/export/nfsroot ip=dhcp rw
  rootwait
  ...
  ...
  ...
  [  151.350209] IP-Config: Complete:
  [  151.353280]      device=eth0, addr=172.17.150.74, mask=255.255.252.0,
gw=172.17.148.1,
  [  151.361127]      host=172.17.150.74, domain=nvidia.com, nis-
domain=(none),
  [  151.368004]      bootserver=0.0.0.0, rootserver=172.17.149.151,
rootpath=
  [  151.480331] VFS: Mounted root (nfs filesystem) on device 0:16.
  [  151.486738] devtmpfs: mounted
```

Appendix III – NFS Boot with Autotest Enabled – HowTo

1) Make sure you have an autotest enable build (--autotest flagged throughout the build process).

2) Depending on whether you are modifying a nightly to NFS Boot or you are using your own build, you may have to run the ./mod_image_for_test.sh script to get it working. If so, enter the chroot and move to the ~/trunk/src/scripts. If you just run ./mod_image_for_test.sh it will modify your latest build, else you can flag the board type and specific image with the "-b" and "-i" flags, respectively.

3) Once you make the modifications as described above, you can boot using the same boot commands and should see a similar boot console output.

4) Login using the username "chronos" and the password "test0000".

5) Make sure that sshd is running and you have an open ssh-server.

6) To run any tests, follow the instructions on https://wiki.nvidia.com/wmpwiki/index.php/Chromium_Seaboard_sqa_wiki#Autotest_test_setup

Appendix IV – DHCP Boot – HowTo

1) In order to run DHCP Boot you must flash the developer version of U-Boot on the test board. The newest version of U-Boot has DHCP/NFS options configured into the developer version so you must first use the nvbuild script to build the package "u-boot-next".

2) You will now have updated versions of U-Boot in your chroot. Flash the developer version of U-Boot to your test board (follow the build steps on how to flash U-Boot to the bootloader). You may need to modify the flashing script to strictly specify "u-boot-developer.bin".

The file you want is "./trunk/chroot/build/tegra2_seaboard/u-boot/u-boot-developer.bin"

3) Follow the above NFS Boot directions through step 9. Make sure you have easy access to your vmlinux.uimg kernel file (with all of the nfs steps configured) because you will need it soon.

4) Set up a DHCP Server on your host machine. This DHCP Server will provide IP addresses to targets on your network.

```
$ sudo apt-get install dhcp3-server
```

Edit /etc/dhcp3/dhcpd.conf and add details about your subnet, including the range of IP addresses you want to give out and any fixed IP addresses you want to allocate for your targets:

```
$ subnet 192.168.4.0 netmask 255.255.255.0 {
$ range 192.168.4.20 192.168.4.50;
$ option routers 192.168.4.1;
$ }
$
$ host seaboard {
$ hardware ethernet 00:23:7d:09:80:0e;
$ fixed-address seaboard0;
$ }
```

(you may want to put seaboard0 in your /etc/hosts file in this example, or you can use a numeric address)

Then start up the server:

```
$ /etc/init.d/dhcp3-server restart
```

5) Test your DHCP Setup as follows:

```
$ CrOS> usb start
$ (Re)start USB...
$ USB:   Tegra ehci init hccr c5008100 and hcor c5008140 hc_length 64
$ Register 10011 NbrPorts 1
```

```
  $ USB EHCI 1.00
  $ scanning bus for devices... 5 USB Device(s) found
  $ scanning bus for storage devices... 1 Storage Device(s) found
  $ scanning bus for ethernet devices... 1 Ethernet Device(s) found
  $ CrOS> bootp
  $ Waiting for Ethernet connection... done.
  $ BOOTP broadcast 1
  $ DHCP client bound to address 172.22.73.81
```

6) Set up a TFTP server on your host. The TFTP server will send a kernel to U-Boot when it asks.

```
  $ sudo apt-get install tftpd-hpa
```

Edit /etc/default/tftpd-hpa like this, replacing the username with your own username (echo $USER):

```
  $ TFTP_USERNAME="nvidia"
  $ TFTP_DIRECTORY="/tftpboot"
  $ TFTP_ADDRESS="0.0.0.0:69"
  $ TFTP_OPTIONS="-v"
```

Now we must copy our modified kernel from the NFS Boot steps into the tftpboot directory so the target can easily read it. Replace the filename with your user, board and serial:

```
  $ sudo mkdir /tftpboot
  $ cd /tftpboot
  $ sudo cp *path to your modified kernel file as mentioned in step 1 -
vmlinux.uimg* uImage-nvidia-seaboard-26
  $ sudo restart tftpd-hpa
```

Test your TFTP server. Ensure that you have an IP address (as shown in the DHCP section above). Then this should read in the kernel:

```
  $ CrOS> tftpboot ${loadaddr} ${tftpserverip}:/tftpboot/uImage-nvidia-
seaboard-26
  $ Waiting for Ethernet connection... done.
  $ Using asx0 device
  $ TFTP from server 172.22.73.60; our IP address is 172.22.73.81
  $ Filename 'uImage-nvidia-seaboard-26'.
  $ Load address: 0x40c000
  $ Loading:
#################################################################
  $ #################################################################
  $ #################################################################
  $ ###############################################
  $ done
  $ Bytes transferred = 3545596 (3619fc hex)
```

7) Next, you must set your board up for full network booting. Replace the IP addresses with the address of your server.

```
  $ setenv serverip 172.22.73.60
  $ setenv tftpserverip 172.22.73.60
  $ setenv nfsserverip 172.22.73.60
  $ setenv board seaboard
  $ setenv serial# 26
  $ setenv user nvidia
```

Certain variables must also be edited from the default boot commands.

```
  $ setenv bootfile uImage-${user}-${board}-${serial#}
  $ setenv tftppath uImage-${user}-${board}-${serial#}
  $ setenv dhcp_setup setenv tftppath uImage-${user}=${board}-${serial#}
  $ setenv regen_net_bootargs setenv bootdev_bootargs dev=/dev/nfs rw
nfsroot=${nfsserverip}:${rootpath} ip=dhcp; run regen_all
```

8) Boot the system by running the following command:

```
  $ CrOS> boot
```

If succesful, you will see a console output similar to the NFS Boot console output shown above.