Optimizing Multiplayer Game Server Performance on AWS

April 2017

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

This whitepaper discusses the exciting use case of running multiplayer game servers in the AWS Cloud and the optimizations that you can make to achieve the highest level of performance. In this whitepaper, we provide you the information you need to take advantage of the Amazon Elastic Compute Cloud (EC2) family of instances to get the peak performance required to successfully run a multiplayer game server on Linux in AWS.

This paper is intended for technical audiences that have experience tuning and optimizing Linux-based servers.

Introduction

Amazon Web Services (AWS) provides benefits for every conceivable gaming workload, including PC/console single and multiplayer games as well as mobile-based, social-based, and web-based games. Running PC/console multiplayer game servers in the AWS Cloud is particularly illustrative of the success and cost reduction that you can achieve with the cloud model over traditional on-premises data centers or colocations.

Multiplayer game servers are based on a client/server network architecture, in which the game server holds the authoritative source of events for all clients (players). Typically, after players send their actions to the server, the server runs a simulation of the game world using all of these actions and sends the results back to each client.

With Amazon Elastic Compute Cloud (Amazon EC2) you can create and run a virtual server (called an *instance*) to host your client/server multiplayer game.[1] Amazon EC2 provides resizable compute capacity and supports Single Root I/O Virtualization (SR-IOV), high frequency processors. For the compute family of instances Amazon EC2 will support up to 72 vCPUs (36 physical cores) when we launch the C5 compute-optimized instance type in 2017.

This whitepaper discusses how to optimize your Amazon EC2 Linux multiplayer game server to achieve the best performance while maintaining scalability, elasticity, and global reach. We start with a brief description of the performance capabilities of the compute optimized instance family and then dive into optimization techniques for networking, CPU, memory, and disk. Finally, we briefly cover benchmarking and testing.

Amazon EC2 Instance Type Considerations

To get the maximum performance out of an Amazon EC2 instance, it is important to look at the compute options available. In this section, we discuss the capabilities of the Amazon EC2 compute optimized instance family that make it ideal for multiplayer game servers.

Amazon EC2 Compute Optimized Instance Capabilities

The current generation C4 compute optimized instance family is ideal for running your multiplayer game server.[2] (The C5 instance type, announced at AWS re:Invent 2016, will be the recommended game server platform when it launches.) C4 instances run on hardware using the Intel Xeon E5-2666 v3 (Haswell) processor. This is a custom processor designed specifically for AWS. The following table lists the capabilities of each instance size in the C4 family.

Instance Size	vCPU Count	RAM (GiB)	Network Performance	EBS Optimized: Max Bandwidth (Mbps)
c4.large	2	3.75	Moderate	500
c4.xlarge	4	7.5	Moderate	750
c4.2xlarge	8	15	High	1000
c4.4xlarge	16	30	High	2000
c4.8xlarge	36	60	10 Gbps	4000

As the table shows, the c4.8xlarge instance provides 36 vCPUs. Since each vCPU is a hyperthread of a full physical CPU core, you get a total of 18 physical cores with this instance size. Each core runs at a base of 2.9 GHz but can run at 3.2 GHz all core turbo (meaning that each core can run simultaneously at 3.2 GHz, even if all the cores are in use) and at a max turbo of 3.5 GHz (possible when only a few cores are in use).

We recommend the c4.4xlarge and c4.8xlarge instance sizes for running your game server because they get exclusive access to one or both of the two underlying processor sockets, respectively. Exclusive access guarantees that you get a 3.2 GHz all core turbo for most workloads. The primary exception is for applications running Advanced Vector Extension (AVX) workloads.[3] If you run AVX workloads on the c4.8xlarge instance, the best you can expect in most cases is 3.1 GHz when running three cores or less. It is important to test your specific workload to verify the performance you can achieve.

The following table shows a comparison between the c4.4xlarge instances and the c4.8xlarge instances for AVX and non-AVX workloads.

C4 Instance Size and Workload	Max Core Turbo Frequency (GHz)	All Core Turbo Frequency (GHz)	Base Frequency (GHz)
C4.8xlarge – non AVX workload	3.5 (when fewer than about 4 vCPUs are active)	3.2	2.9
C4.8xlarge – AVX workload	≤ 3.3	≤ 3.1 depending on the workload and number of active cores	2.5
C4.4xlarge – non AVX workload	3.2	3.2	2.9
C4.4xlarge – AVX workload	3.2	≤ 3.1 depending on the workload and number of active cores	2.5

Alternative Compute Instance Options

There are situations, for example, for some role-playing games (RPGs) and multiplayer online battle arenas (MOBAs), where your game server can be more memory bound than compute bound. In these cases, the M4 instance type may be a better option than the C4 instance type since it has a higher memory to vCPU ratio. The compute optimized instance family has a higher vCPU to memory ratio than other instance families while the M4 instance has a higher memory to vCPU ratio. M4 instances use a Haswell processor for the m4.10xlarge and m4.16xlarge sizes; smaller sizes use either a Broadwell or a Haswell processor. The M4 instance type is similar to the C4 instance type in networking performance and has plenty of bandwidth for game servers.

Performance Optimization

There are many performance options for Linux servers, with networking and CPU being the two most important. This section documents the performance options that AWS gaming customers have found the most valuable and/or the options that are the most appropriate for running game servers on virtual machines (VMs).

The performance options are categorized into four sections: networking, CPU, memory, and disk. This is not an all-inclusive list of performance tuning options, and not all of the options will be appropriate for every gaming workload. We strongly recommend testing these settings before implementing them in production.

This section assumes that you are running your instance in a VPC created with Amazon Virtual Private Cloud (VPC)[4] that uses an Amazon Machine Image (AMI)[5] with a hardware virtual machine (HVM). All of the instructions and settings that follow have been verified on the Amazon Linux AMI 2016.09 using the 4.4.23-31.54 kernel, but they should work with all future releases of Amazon Linux.

Networking

Networking is one of the most important areas for performance tuning. Multiplayer client/server games are extremely sensitive to latency and dropped packets. A list of performance tuning options for networking is provided in the following table.

Performance Tuning Option	Summary	Notes	Links or Commands
Deploying game servers close to players	Proximity to players is the best way to reduce latency	AWS has numerous Regions across the globe.	List of AWS Regions
Enhanced networking	Improved networking performance	Nearly every workload should benefit. No downside.	Linux/Windows
UDP			
Receive buffers	Helps prevent dropped packets	Useful when the latency between client and server is high. Little downside but should be tested.	Add the following to /etc/sysctl.conf: net.core.rmem_default = New_Value net.core.rmem_max = New_Value (Recommend start by doubling the current values set for your system)
Busy polling	Reduce latency of incoming packet processing	Can increase CPU utilization	Add the following to /etc/sysctl.conf: net.core.busy_read = New_Value net.core.busy_poll = New_Value (Recommend testing a value of 50 first then 100)

Performance Tuning Option	Summary	Notes	Links or Commands
Memory	Helps prevent dropped packets		Add the following to /etc/sysctl.conf: net.ipv4.udp_mem = New_Value New_Value New_Value (Recommend doubling the current values set for your system)
Backlog	Helps prevent dropped packets		Add the following to /etc/sysctl.conf: net.core.netdev_max_backlog= New_Value (Recommend doubling the current values set for your system)
Transmit and receive queues	Possible performance boost by disabling hyperthreading		

The following recommendations cover how to reduce latency, avoid dropped packets, and obtain optimal networking performance for your game servers.

Deploying Game Servers Close to Players

Deploying your game servers as close as possible to your players is a key element for good player experience. AWS has numerous Regions across the world, which allows you to deploy your game servers close to your players. For the most current list of AWS Regions and Availability Zones, see https://aws.amazon.com/about-aws/global-infrastructure/.[6]

You can package your instance AMI and deploy it to as many Regions as you choose. Customers often deploy AAA PC/console games in almost every available Region. As you determine where your players are globally you can decide where to deploy your game servers to provide the best experience possible.

Enhanced Networking

Enhanced networking is another performance tuning option.[7] Enhanced networking uses single root I/O virtualization (SR-IOV) and exposes the

network card directly to the instance without needing to go through the hypervisor.[8] This allows for generally higher I/O performance, lower CPU utilization, higher packets per second (PPS) performance, lower inter-instance latencies, and very low network jitter. The performance improvement provided by enhanced networking can make a big difference for a multiplayer game server.

Enhanced networking is only available for instances running in a VPC using an HVM AMI and only for certain instance types, such as the C4, R4, R3, I3, I2, M4, and D2. These instance types use the Intel 82599 Virtual Function Interface (which uses the "ixgbevf" Linux driver.) In addition, the X1, R4, P2, and M4.16xlarge (and soon the C5) instances support enhanced networking using the Elastic Network Adapter (ENA).

The Amazon Linux AMI includes these necessary drivers by default. Follow the Linux or Windows instructions to install the driver for other AMIs.[9, 10] It is important to have the latest ixgbevf driver, which can be downloaded from Intel's website.[11] The minimum recommended version for the ixgbevf driver is version 2.14.2.

To check the driver version running on your instance run the following command:

```
ethtool -i eth0
```

User Datagram Protocol (UDP)

Most first-person shooter games and other similar client/server multiplayer games use UDP as the protocol for communication between clients and game servers. The following sections lay out four UDP optimizations that can improve performance and reduce the occurrence of dropped packets.

Receive Buffers

The first UDP optimization is to increase the default value for the receive buffers. Having too little UDP buffer space can cause the operating system kernel to discard UDP packets, resulting in packet loss. Increasing this buffer space can be helpful in situations where the latency between the client and server is high. The default value for both `rmem_default` and `rmem_max` on Amazon Linux is 212992.

To see the current default values for your system run the following commands:

```
cat /proc/sys/net/core/rmem_default
cat /proc/sys/net/core/rmem_max
```

A common approach to allocating the right amount of buffer space is to first double both values and then test the performance difference this makes for your game server. Depending on the results, you may need to decrease or increase these values. Note that the `rmem_default` value should not exceed the `rmem_max` value.

To configure these parameters to persist across reboots set the new `rmem_default` and `rmem_max` values in the /etc/sysctl.conf file:

```
net.core.rmem_default = New_Value
net.core.rmem_max = New_Value
```

Whenever making changes to the sysctl.conf file you should run the following command to refresh the configuration:

```
sudo sysctl -p
```

Busy Polling

A second UDP optimization is busy polling, which can help reduce network receive path latency by having the kernel poll for incoming packets. This will increase CPU utilization but can reduce delays in packet processing.

On most Linux distributions, including Amazon Linux, busy polling is disabled by default. We recommend that you start with a value of 50 for both `busy_read` and `busy_poll` and then test what difference this makes for your game server. `Busy_read` is the number of microseconds to wait for packets on the device queue for socket reads, while `busy_poll` is the number of microseconds to wait for packets on the device queue for socket poll and selects. Depending on the results, you may need to increase the value to 100.

To configure these parameters to persist across reboots add the new `busy_read` and `busy_poll` values to the /etc/sysctl.conf file:

```
net.core.busy_read = New_Value
net.core.busy_poll = New_Value
```

Again, run the following command to refresh the configuration after making changes to the sysctl.conf file:

```
sudo sysctl -p
```

UDP Buffers

A third UDP optimization is to change how much memory the UDP buffers use for queueing. The `udp_mem` option configures the number of pages the UDP sockets can use for queueing. This can help reduce dropped packets when the network adaptor is very busy.

This setting is a vector of three values that are measured in units of pages (4096 bytes). The first value, called *min*, is the minimum threshold before UDP moderates memory usage. The second value, called *pressure*, is the memory threshold after which UDP will moderate the memory consumption. The final value, called *max*, is the maximum number of pages available for queueing by all UDP sockets. By default, Amazon Linux on the c4.8xlarge instance uses a vector of `1445727 1927636 2891454`, while the c4.4xlarge instance uses a vector of `720660 960882 1441320`.

To see the current default values run the following command:

```
cat /proc/sys/net/ipv4/udp_mem
```

A good first step when experimenting with new values for this setting is to double the values and then test what difference this makes for your game server. It is also good to adjust the values so they are multiples of the page size (4096 bytes). To configure these parameters to persist across reboots add the new UDP buffer values to the /etc/sysctl.conf file:

```
net.ipv4.udp_mem = New_Value New_Value New_Value
```

Run the following command to refresh the configuration after making changes to the sysctl.conf file:

```
sudo sysctl -p
```

Backlog

The final UDP optimization that can help reduce the chance of dropped packets is to increase the backlog value. This optimization will increase the queue size for incoming packets for situations where the interface is receiving packets at a faster rate than the kernel can handle. On Amazon Linux the default value of the queue size is 1000.

To check the default value run the following command:

```
cat /proc/sys/net/core/netdev_max_backlog
```

We recommend that you double the default value for your system and then test what difference this makes for your game server. To configure these parameters to persist across reboots add the new `backlog` value to the /etc/sysctl.conf file:

```
net.core.netdev_max_backlog = New_Value
```

Run the following command to refresh the configuration after making changes to the sysctl.conf file:

```
sudo sysctl -p
```

Transmit and Receive Queues

Many game servers put more pressure on the network through the number of packets per second being processed rather than on the overall bandwidth used.

In addition, I/O wait can become a bottleneck if one of the vCPUs gets a large volume of interrupt requests (IRQs).

Receive Side Scaling (RSS) is a common method used to address these networking performance issues.[12] RSS is a hardware option that can provide multiple receive queues on a network interface controller (NIC). For Amazon Elastic Compute Cloud (Amazon EC2), the NIC is called an Elastic Network Interface (ENI).[13] RSS is enabled on the C4 instance family but changes to the configuration of RSS are not allowed. The C4 instance family provides two receive queues for all of the instance sizes when using Linux. Each of these queues has a separate IRQ number and is mapped to a separate vCPU.

Running the command `$ ls -1 /sys/class/net/eth0/queues` on a c4.8xlarge instance displays the following queues:

```
$ ls -l /sys/class/net/eth0/queues
total 0
drwxr-xr-x 2 root 0 Aug 18 21:00 rx-0
drwxr-xr-x 2 root root 0 Aug 18 21:00 rx-1
drwxr-xr-x 3 root root 0 Aug 18 21:00 tx-0
drwxr-xr-x 3 root root 0 Aug 18 21:00 tx-1
```

To find out which IRQs are being used by the queues and how the CPU is handling those interrupts run the following command:

```
cat /proc/interrupts
```

Alternatively, run this command to output the IRQs for the queues:

```
echo eth0; grep eth0-TxRx /proc/interrupts | awk '{printf "
%s\n", $1}'
```

What follows is the reduced output when viewing the full contents of /proc/interrupts on a c4.8xlarge instance showing just the eth0 interrupts. The first column is the IRQ for each queue. The last two columns are the process

information. In this case, you can see the TxRx-0 and TxRx-1 are using IRQs 267 and 268, respectively.

```
     CPU0  CPU23  CPU33
267  634   2789   0      xen-pirq-msi-x  eth0-TxRx-0
268  600   0      2587   xen-pirq-msi-x  eth0-TxRx-1
```

To verify which vCPU the queue is sending interrupts to run the following commands (replacing IRQ_Number with the IRQ for each TxRx queue):

```
$ cat /proc/irq/267/smp_affinity
00000000,00000000,00000000,00800000
$ cat /proc/irq/268/smp_affinity
00000000,00000000,00000002,00000000
```

The previous output is from a c4.8xlarge instance. It is in hex and needs to be converted to binary to find the vCPU number. For example, the hex value 00800000 converted to binary is 00000000100000000000000000000000. Counting from the right and starting at 0 you get to vCPU 23. The other queue is using vCPU 33.

Because vCPUs 23 and 33 are on different processor sockets, they are physically on different non-uniform memory access (NUMA) nodes. One issue here is that each vCPU is, by default, a hyperthread (but in this particular case they are each hyperthreads of the same core), so a performance boost could be seen by tying each queue to a physical core.

The IRQs for the two queues on Amazon Linux on the C4 instance family are already pinned to particular vCPUs that are on separate NUMA nodes on the c4.8xlarge instance. This default state may be ideal for your game servers. However, it is important to verify on your distribution of Linux that there are two queues that are configured for IRQs and vCPUs (which are on separate NUMA nodes). On C4 instance sizes other than the c4.8xlarge, NUMA is not an issue since the other sizes only have one NUMA node.

One option that could improve performance for RSS is to disable hyperthreading. If you disable hyperthreading on Amazon Linux, then, by

default, the queues will be pinned to physical cores (which will also be on separate NUMA nodes on the c4.8xlarge instance). See the Hyperthreading section in this whitepaper for more information on how to disable hyperthreading.

If you don't pin game server processes to cores, you could prevent the Linux scheduler from assigning game server processes to the vCPUs (or cores) for the RSS queues. To do this you need to configure two options.

First, in your text editor, edit the /boot/grub/grub.conf file. For the first entry that begins with "kernel" (there may be more than one kernel entry, you only need to edit the first one), add `isolcpus=NUMBER` at the end of the line, where `NUMBER` is the number of the vCPUs for the RSS queues. For example, if the queues are using vCPUs 3 and 4, replace `NUMBER` with "3-4".

```
# created by imagebuilder
default=0
timeout=1
hiddenmenu
title Amazon Linux 2014.09 (3.14.26-24.46.amzn1.x86_64)
root (hd0,0)
kernel /boot/vmlinuz-3.14.26-24.46.amzn1.x86_64 root=LABEL=/
console=ttyS0 isolcpus=NUMBER
initrd /boot/initramfs-3.14.26-24.46.amzn1.x86_64.img
```

Using isolcpus will prevent the scheduler from running the game server processes on the vCPUs you specify. The problem is that it will also prevent irqbalance from assigning IRQs to these vCPUs. To fix this you need to use the IRQBALANCE_BANNED_CPUS option to ban all of the remaining CPUs. Version 1.1.10 or later of irqbalance on current versions of Amazon Linux prefers the IRQBALANCE_BANNED_CPUS option and will assign IRQs to the vCPUs specified in isolcpus in order to honor the vCPUs specified by IRQBALANCE_BANNED_CPUS. Therefore, for example, if you isolated vCPUs 3-4 using isolcpus, you would then need to ban the other vCPUs on the instance using IRQBALANCE_BANNED_CPUS.

To do this you need to use the IRQBALANCE_BANNED_CPUS option in the /etc/sysconfig/irqbalance file. This is a 64-bit hexadecimal bit mask. The best way to find the value would be to write out the vCPUs you want to include in

this value in decimal format and then convert to hex. So in the earlier example where we used isolcpus to exclude vCPUs 3-4, we would then want to use IRQBALANCE_BANNED_CPUS to exclude vCPUs 1, 2, and 5-14 (assuming we are on a c4.4xlarge instance), which would be 1111111111100111 in decimal and finally FFE7n when converted to hex. Add the following line to the /etc/sysconfig/irqbalance file using your favorite editor:

```
IRQBALANCE_BANNED_CPUS="FFE7n"
```

The result is that vCPUs 3 and 4 will not be used by the game server processes but will be used by the RSS queues and a few other IRQs used by the system.

Like everything else, all of these values should be tested with your game server to determine what the performance difference is.

Bandwidth

The C4 instance family offers plenty of bandwidth for a multiplayer game server. The c4.4xlarge instance provides high network performance, and up to 10 Gbps is achievable between two c4.8xlarge instances (or other large instance sizes like the m4.10xlarge) that are using enhanced networking and are in the same placement group.[14] The bandwidth provided by both the c4.4xlarge and c4.8xlarge instances has been more than sufficient for every game server use case we have seen.

You can easily determine the networking performance for your workload on a C4 instance compared to other instances in the same Availability Zone, other instances in another Availability Zone, and most importantly, to and from the Internet. Iperf is probably one of the best tools for determining network performance on Linux,[15] while Nttcp is a good tool for Windows.[16] The previous links also provide instructions on doing network performance testing. Outside of the placement group, you need to use a tool like Iperf or Nttcp to determine the exact network performance achievable for your game server.

CPU

CPU is one of the two most important performance-tuning areas for game servers.

Performance Tuning Option	Summary	Notes	Links or Commands
Clock Source	Using tsc as the clock source can improve performance for game servers	Xen is the default clocksource on Amazon Linux.	Add the following entry to the kernel line of the /boot/grub/grub.conf file: tsc=reliable clocksource=tsc
C-State and P-State	C-state and P-state options are optimized by default, except for the C-state on the c4.8xlarge. Setting C-state to C1 on the c4.8xlarge should improve CPU performance.	Can only be changed on the c4.8xlarge. Downside is that 3.5 GHz max turbo will not be available. However, the 3.2 GHz all core turbo will be available.	Add the following entry to the kernel line of the /boot/grub/grub.conf file: intel_idle.max_cstate=1
Irqbalance	When not pinning game servers to vCPUs irqbalance can help improve CPU performance.	Installed and running by default on Amazon Linux. Check your distribution to see if this is running.	NA
Hyperthreading	Each vCPU is a hyperthread of a core. Performance may improve by disablinghyperthreading.		Add the following entry to the kernel line of the /boot/grub/grub.conf file: Maxcpus=X (where X is the number of actual cores in the instance)
CPU Pinning	Pinning the game server process to vCPU can provide benefits in some situations.	CPU pinning does not appear to be a common practice among game companies.	"numactl --physcpubind $phys_cpu_core --membind $associated_numa_node ./game_server_executable"
Linux Scheduler	There are three particular Linux scheduler configuration options that can help with game servers.		sudo sysctl -w 'kernel.sched_min_granularity_ns=New_Value' (Recommend start by doubling the current value set for your system) sudo sysctl -w 'kernel.sched_wakeup_granularity_ns=New_Value' sudo sysctril –w (Recommend start by halving the current value set for your system) 'kernel.sched_migration_cost_ns=New_Value'

Performance Tuning Option	Summary	Notes	Links or Commands
			(Recommend start by doubling the current value set for your system)

Clock Source

A clock source gives Linux access to a timeline so that a process can determine where it is in time. Time is extremely important when it comes to multiplayer game servers given that the server is the authoritative source of events and yet each client has its own view of time and the flow of events. The kernel.org web site has a good introduction to clock sources.[17]

To find the current clock source:

```
$cat
/sys/devices/system/clocksource/clocksource0/current_clocksource
```

By default, on a C4 instance running Amazon Linux this is set to xen.

To view the available clock sources:

```
cat
/sys/devices/system/clocksource/clocksource0/available_clocksource
```

This list should show xen, tsc, hpet, and acpi_pm by default on a C4 instance running Amazon Linux. For most game servers the best clock source option is TSC (Time Stamp Counter), which is a 64-bit register on each processor. In most cases, TSC is the fastest, highest-precision measurement of the passage of time and is monotonic and invariant. See this xen.org article for a good discussion about TSC when it comes to XEN virtualization.[18] Synchronization is provided across all processors in all power states so TSC is considered synchronized and invariant. This means that TSC will increment at a constant rate.

TSC can be accessed using the rdtsc or rdtscp instructions. Rdtscp is often a better option than rdtsc since rdtscp takes into account that Intel processors

sometimes use out-of-order execution, which can affect getting accurate time readings.

The recommendation for game servers is to change the clock source to TSC. However, you should test this thoroughly for your workloads. To set the clock source to TSC, edit the /boot/grub/grub.conf file with your editor of choice. For the first entry that begins with "kernel" (note that there may be more than one kernel entry, you only need to edit the first one), add `tsc=reliable clocksource=tsc` at the end of the line.

```
# created by imagebuilder
default=0
timeout=1
hiddenmenu
title Amazon Linux 2014.09 (3.14.26-24.46.amzn1.x86_64)
root (hd0,0)
kernel /boot/vmlinuz-3.14.26-24.46.amzn1.x86_64 root=LABEL=/
console=ttyS0 tsc=reliable clocksource=tsc
initrd /boot/initramfs-3.14.26-24.46.amzn1.x86_64.img
```

Processor State Control (C-States and P-States)

Processor State Controls can only be modified on the c4.8xlarge instance (also configurable on the d2.8xlarge, m4.10xlarge, and x1.32xlarge instances).[19] C-states control the sleep levels that a core can enter when it is idle, while P-states control the desired performance (in CPU frequency) for a core. C-states are idle power saving states, while P-states are execution power saving states.

C-states start at C0, which is the shallowest state where the core is actually executing functions, and go to C6, which is the deepest state where the core is essentially powered off. The default C-state for the c4.8xlarge instance is C6. For all of the other instance sizes in the C4 family the default is C1. This is the reason that the 3.5 GHz max turbo frequency is only available on the c4.8xlarge instance. Some vCPUs need to be in a deeper sleep state than C1 in order for the cores to hit 3.5 GHz.

An option on the c4.8xlarge instance is to set C1 as the deepest C-state to prevent the cores from going to sleep. That reduces the processor reaction latency but also prevents the cores from hitting the 3.5 GHz Turbo Boost if only a few cores are active; it would still allow the 3.2 GHz all core turbo. Therefore,

you would be trading the possibility of achieving 3.5 GHz when a few cores are running for the reduced reaction latency. Your results will depend on your testing and application workloads. If 3.2 GHz all core turbo is acceptable and you plan to utilize all or most of the cores on the C4.8xlarge instance, then change the C-state to C1.

P-states start at P0, where Turbo mode is enabled, and go to P15, which represents the lowest possible frequency. P0 provides the maximum baseline frequency. The default P-state for all C4 instance sizes is P0. There is really no reason for changing this for gaming workloads. Turbo Boost mode is the desirable state.

The following table describes the C- and P-states for the c4.4xlarge and c4.8xlarge.

Instance size	Default Max C-State	Recommended setting	Default P-State	Recommended setting
c4.4xlarge and smaller	1	1	0	0
c4.8xlarge	6[a]	1	0	0

a) Running cat /sys/module/intel_idle/parameters/max_cstate will show the max C-state as 9. It is actually set to 6, which is the maximum possible value.

Use turbostat to see the C-state and max turbo frequency that can be achieved on the c4.8xlarge instance. Again, these instructions were tested using the Amazon Linux AMI and only work on the c4.8xlarge instance but not on any of the other instance sizes in the C4 family.

First, run the following turbostat command to install stress on your system. (If turbostat is not installed on your system then install that, too.)

```
sudo yum install stress
```

The following command stresses two cores (i.e., two hyperthreads of two different physical cores):

```
sudo turbostat --debug stress -c 2 -t 60
```

Here is a truncated printout of the results of running the command:

Core	CPU	Avg_MHz	%Busy	Bzy_MHz	TSC_MHz	SMI	CPU%c1	CPU%c3	CPU%c6
-	-	188	5.50	3428	2893	0	9.82	0.00	84.68
0	0	4	0.11	3259	2893	0	4.48	0.01	95.40
0	18	4	0.12	3274	2893	0	4.47		
1	1	4	0.11	3270	2893	0	4.45	0.00	95.44
1	19	4	0.11	3280	2893	0	4.45		
2	2	2	0.06	3408	2893	0	99.93	0.00	0.01
2	20	3327	97.20	3431	2893	0	2.79		
3	3	4	0.11	3278	2893	0	4.45	0.00	95.44
3	21	4	0.11	3279	2893	0	4.45		
4	4	3	0.11	3276	2893	0	3.57	0.00	96.32
4	22	4	0.11	3276	2893	0	3.57		
5	5	3	0.10	3277	2893	0	4.50	0.00	95.40
5	23	4	0.11	3279	2893	0	4.49		
6	6	4	0.11	3277	2893	0	4.45	0.00	95.44
6	24	3	0.10	3280	2893	0	4.45		
7	7	4	0.11	3279	2893	0	3.57	0.00	96.32
7	25	4	0.11	3280	2893	0	3.57		
8	8	4	0.11	3266	2893	0	7.09	0.00	92.80
8	26	4	0.12	3276	2893	0	7.08		
0	9	3324	97.13	3430	2893	0	2.86	0.00	0.01
0	27	2	0.05	3407	2893	0	99.94		
1	10	4	0.11	3276	2893	0	4.47	0.00	95.42
1	28	3	0.11	3277	2893	0	4.47		
2	11	4	0.11	3268	2893	0	4.46	0.00	95.43
2	29	4	0.11	3265	2893	0	4.45		
3	12	3	0.11	3270	2893	0	4.46	0.00	95.43
3	30	4	0.11	3270	2893	0	4.46		
4	13	4	0.14	3275	2893	0	3.59	0.00	96.27
4	31	3	0.11	3280	2893	0	3.62		
5	14	4	0.12	3277	2893	0	4.45	0.00	95.43
5	32	3	0.10	3278	2893	0	4.47		
6	15	3	0.11	3276	2893	0	7.11	0.00	92.79
6	33	4	0.12	3264	2893	0	7.09		
7	16	4	0.11	3276	2893	0	4.49	0.00	95.40
7	34	4	0.11	3280	2893	0	4.48		
8	17	4	0.12	3272	2893	0	4.45	0.00	95.43
8	35	3	0.11	3272	2893	0	4.46		

Definitions:

AVG_MHz: number of cycles executed divided by time elapsed.

%Busy: percent of time in "C0" state.

Bzy_MHz: average clock rate while the CPU was busy (in "c0" state).

TSC_MHz: average MHz that the TSC ran during the entire interval.

The output shows that vCPUs 9 and 20 spent most of the time in the C0 state (%Busy) and hit close to the maximum turbo of 3.5 GHz (Bzy_MHz). vCPUs 2 and 27, the other hyperthreads of these cores, are sitting in C1 C-state (CPU%c1) waiting for instructions. A frequency close to 3.5 GHz was achievable because the default C-state on the c4.8xlarge instance was C6, and so most of the cores were in the C6 state (CPU%c6).

Next, try stressing all 36 vCPUs to see the 3.2 GHz All Core Turbo:

```
sudo turbostat --debug stress -c 36 -t 60
```

Here is a truncated printout of the results of running the command:

Core	CPU	Avg_MHz	%Busy	Bzy_MHz	TSC_MHz	SMI	CPU%c1	CPU%c3	CPU%c6
-	-	3189	99.90	3200	2893	0	0.06	0.00	0.05
0	0	3188	99.85	3200	2893	0	0.13	0.00	0.02
0	18	3192	99.98	3200	2893	0	0.01		
1	1	3191	99.94	3200	2893	0	0.01	0.00	0.05
1	19	3188	99.86	3200	2893	0	0.09		
2	2	3191	99.95	3200	2893	0	0.01	0.00	0.04
2	20	3187	99.81	3200	2893	0	0.15		
3	3	3189	99.88	3200	2893	0	0.09	0.00	0.03
3	21	3191	99.96	3200	2893	0	0.01		
4	4	3192	99.99	3200	2893	0	0.00	0.00	0.01
4	22	3189	99.90	3200	2893	0	0.09		
5	5	3190	99.93	3200	2893	0	0.01	0.00	0.06
5	23	3187	99.81	3200	2893	0	0.13		
6	6	3191	99.97	3200	2893	0	0.01	0.00	0.02
6	24	3189	99.89	3200	2893	0	0.09		
7	7	3187	99.83	3200	2893	0	0.09	0.00	0.08
7	25	3190	99.91	3200	2893	0	0.01		
8	8	3187	99.84	3200	2893	0	0.09	0.00	0.07
8	26	3190	99.92	3200	2893	0	0.01		
0	9	3188	99.84	3200	2893	0	0.08	0.00	0.07
0	27	3190	99.91	3200	2893	0	0.01		
1	10	3188	99.84	3200	2893	0	0.09	0.00	0.07
1	28	3190	99.92	3200	2893	0	0.01		
2	11	3188	99.85	3200	2893	0	0.09	0.00	0.06
2	29	3190	99.93	3200	2893	0	0.01		
3	12	3188	99.86	3200	2893	0	0.09	0.00	0.05
3	30	3191	99.94	3200	2893	0	0.01		
4	13	3191	99.95	3200	2893	0	0.01	0.00	0.04
4	31	3186	99.81	3200	2893	0	0.15		
5	14	3192	99.98	3200	2893	0	0.00	0.00	0.01
5	32	3189	99.89	3200	2893	0	0.09		
6	15	3189	99.88	3200	2893	0	0.09	0.00	0.03
6	33	3191	99.96	3200	2893	0	0.01		
7	16	3187	99.82	3200	2893	0	0.09	0.00	0.09
7	34	3189	99.90	3200	2893	0	0.01		
8	17	3192	99.97	3200	2893	0	0.01	0.00	0.02
8	35	3185	99.75	3200	2893	0	0.23		

You can see that all of the vCPUs are in C0 for over 99% of the time (%Busy) and that they are all hitting 3.2 GHz (Bzy_MHz) when in C0.

To set the C-State to C1, edit the /boot/grub/grub.conf file with your editor of choice. For the first entry that begins with “kernel”, (there may be more than one kernel entry, you only need to edit the first one) add `intel_idle.max_cstate=1` at the end of the line to set C1 as the deepest C-state for idle cores:

```
# created by imagebuilder
default=0
timeout=1
hiddenmenu
title Amazon Linux 2014.09 (3.14.26-24.46.amzn1.x86_64)
root (hd0,0)
kernel /boot/vmlinuz-3.14.26-24.46.amzn1.x86_64 root=LABEL=/
console=ttyS0 intel_idle.max_cstate=1
initrd /boot/initramfs-3.14.26-24.46.amzn1.x86_64.img
```

Save the file and exit your editor. Reboot your instance to enable the new kernel option. Now rerun the turbostat command to see what changed after setting the C-state to C1:

```
sudo turbostat --debug stress -c 2 -t 10
```

Here is a truncated printout of the results of running the command:

Core	CPU	Avg_MHz	%Busy	Bzy_MHz	TSC_MHz	SMI	CPU%c1	CPU%c3	CPU%c6
-	-	178	5.59	3200	2893	0	94.41	0.00	0.00
0	0	1	0.03	3201	2893	0	99.97	0.00	0.00
0	18	1	0.03	3200	2893	0	99.97		
1	1	1	0.03	3201	2893	0	99.97	0.00	0.00
1	19	1	0.03	3201	2893	0	99.97		
2	2	2	0.05	3200	2893	0	99.95	0.00	0.00
2	20	3192	99.99	3200	2893	0	0.01		
3	3	2	0.06	3201	2893	0	99.94	0.00	0.00
3	21	1	0.04	3201	2893	0	99.96		
4	4	1	0.03	3202	2893	0	99.97	0.00	0.00
4	22	1	0.04	3200	2893	0	99.96		
5	5	1	0.04	3201	2893	0	99.96	0.00	0.00
5	23	1	0.03	3200	2893	0	99.97		
6	6	1	0.04	3201	2893	0	99.96	0.00	0.00
6	24	1	0.04	3200	2893	0	99.96		
7	7	1	0.03	3201	2893	0	99.97	0.00	0.00
7	25	1	0.04	3200	2893	0	99.96		
8	8	1	0.03	3201	2893	0	99.97	0.00	0.00
8	26	1	0.03	3200	2893	0	99.97		
0	9	1	0.04	3201	2893	0	99.96	0.00	0.00
0	27	1	0.03	3200	2893	0	99.97		
1	10	1	0.03	3201	2893	0	99.97	0.00	0.00
1	28	1	0.04	3200	2893	0	99.96		
2	11	1	0.03	3201	2893	0	99.97	0.00	0.00
2	29	1	0.04	3200	2893	0	99.96		
3	12	1	0.03	3201	2893	0	99.97	0.00	0.00
3	30	1	0.03	3201	2893	0	99.97		
4	13	1	0.04	3201	2893	0	99.96	0.00	0.00
4	31	1	0.03	3201	2893	0	99.97		
5	14	1	0.04	3201	2893	0	99.96	0.00	0.00
5	32	1	0.04	3200	2893	0	99.96		
6	15	1	0.03	3202	2893	0	99.97	0.00	0.00
6	33	1	0.04	3201	2893	0	99.96		
7	16	3192	99.99	3200	2893	0	0.01	0.00	0.00
7	34	1	0.04	3200	2893	0	99.96		
8	17	1	0.03	3201	2893	0	99.97	0.00	0.00
8	35	1	0.03	3201	2893	0	99.97		

The output in the table above shows that all of the cores are now at a C-state of C1. The maximum average frequency of the two vCPUs that were stressed, vCPUs 16 and 2 in the example above, is 3.2 GHz (Bzy_MHz). The maximum turbo of 3.5 GHz is no longer available since all of the vCPUs are at C1.

Another way to verify that the C-state is set to C1 is to run the following command:

```
cat /sys/module/intel_idle/parameters/max_cstate
```

Finally, you may be wondering what the performance cost is when a core switches from C6 to C1. You can query the cpuidle file to show the exit latency, in microseconds, for various C-states. There is a latency penalty each time the CPU transitions between C-states.

In the default C-state, cpuidle shows that to move from C6 to C0 requires 133 microseconds:

```
$ find /sys/devices/system/cpu/cpu0/cpuidle -name latency -o -
name name | xargs cat
POLL
0
C1-HSW
2
C1E-HSW
10
C3-HSW
33
C6-HSW
133
```

After you change the C-state default to C1, you can see the difference in CPU idle. Now we see that to move from C1 to C0 takes only 2 microseconds. We have cut the latency by 131 microseconds by setting the vCPUs to C1.

```
$ find /sys/devices/system/cpu/cpu0/cpuidle -name latency -o -
name name | xargs cat
POLL
0
C1-HSW
2
```

The instructions above are only relevant for the c4.8xlarge instance. For the c4.4xlarge instance (and smaller instance sizes in the C4 family), the C-state is already at C1 and all core turbo 3.2 GHz is available by default. Turbostat will not show that the processors are exceeding the base of 2.9 GHz. One problem is that even when using the debug option for turbostat the c4.4xlarge instance does not show the Avg_MHz or the Bzy_MHz values like in the output shown above for the c4.8xlarge instance.

One way to verify that the vCPUs on the c4.4xlarge instance are hitting the 3.2 GHz all core turbo is to use the showboost script from Brendan Gregg.[20]

For this to work on Amazon Linux you need to install the msr tools. To do this run these commands:

```
sudo yum groupinstall "Development Tools"
wget https://launchpad.net/ubuntu/+archive/primary/+files/msr-tools_1.3.orig.tar.gz
tar –zxvf msr-tools_1.3.orig.tar.gz
sudo make
sudo make install
cd msr-tools_1.3
wget https://raw.githubusercontent.com/brendangregg/msr-cloud-tools/master/showboost
chmod +x showboost
sudo ./showboost
```

The output only shows vCPU 0 but you can modify the options section to change the vCPU that will be displayed. To show the CPU frequency run your game server or use turbostat stress, and then run the showboost command to view the frequency for a vCPU.

Irqbalance

Irqbalance is a service that distributes interrupts over the cores in the system to improve performance. Irqbalance is recommended for most use cases except where you are pinning game servers to specific vCPUs or cores. In that case, disabling irqbalance may make sense. Please test this with your specific workloads to see if there is a difference. By default irqbalance is running on the C4 instance family.

To check if irqbalance is running on your instance run the following command:

```
sudo service irqbalance status
```

Irqbalance can be configured in the /etc/sysconfig/irqbalance file.

You want to see a fairly even distribution of interrupts across all the vCPUs. You can view the status of interrupts to see if they are properly being distributed across vCPUs by running the following command:

```
cat /proc/interrupts
```

Hyperthreading

Each vCPU on the C4 instance family is a hyperthread of a physical core. Hyperthreading can be disabled if you determine that this has a detrimental impact on the performance of your application. However, many gaming customers do not find a need to disable hyperthreading.

The table below shows the number of physical cores in each C4 instance size.

Instance Name	vCPU Count	Physical Core Count
c4.large	2	1
c4.xlarge	4	2
c4.2xlarge	8	4
c4.4xlarge	16	8
c4.8xlarge	36	18

All of the vCPUs can be viewed by running the following:

```
cat /proc/cpuinfo
```

To get more specific output you can use the following:

```
egrep '(processor|model name|cpu MHz|physical id|siblings|core
id|cpu cores)' /proc/cpuinfo
```

In this output, the “processor” is the vCPU number. The “physical id” shows the processor socket ID. For any C4 instance other than the c4.8xlarge this will be 0. The “core id” is the physical core number. Each entry that has the same “physical id” and “core id” will be hyperthreads of the same core.

Another way to view the vCPUs pairs (i.e., hyperthreads) of each core is to look at the thread_siblings_list for each core. This will show two numbers that are

the vCPUs for each core. Change the X in "cpuX" to the vCPU number that you want to view.

```
cat /sys/devices/system/cpu/cpuX/topology/thread_siblings_list
```

To disable hyperthreading, edit the /boot/grub/grub.conf file with your editor of choice. For the first entry that begins with "kernel" (there may be more than one kernel entry, you only need to edit the first one), add `maxcpus=NUMBER` at the end of the line, where `NUMBER` is the number of actual cores in the C4 instance size you are using. Refer to the table above on the number of physical cores in each C4 instance size.

```
# created by imagebuilder
default=0
timeout=1
hiddenmenu
title Amazon Linux 2014.09 (3.14.26-24.46.amzn1.x86_64)
root (hd0,0)
kernel /boot/vmlinuz-3.14.26-24.46.amzn1.x86_64 root=LABEL=/
console=ttyS0 maxcpus=18
initrd /boot/initramfs-3.14.26-24.46.amzn1.x86_64.img
```

Save the file and exit your editor. Reboot your instance to enable the new kernel option.

Again, this is one of those settings that you should test to determine if it provides a performance boost for your game. This setting would likely need to be combined with CPU pinning before it would provide any performance boost. In fact, disabling hyperthreading without using pinning may degrade performance. Many major AAA games running on AWS do not actually disable hyperthreading. If there is no performance boost you can avoid this setting to avoid the administrative overhead of having to maintain this on each of your game servers.

CPU Pinning

Many of the game server processes we see usually have a main thread and then a few ancillary threads. Pinning the process for each game server to a core

(either a vCPU or physical core) is definitely an option but not a configuration we often see. Usually pinning is done in situations where the game engine truly needs exclusive access to a core. Often game companies simply allow the Linux scheduler to handle this. Again, this is something that should be tested, but if the performance is sufficient without pinning it can save you administrative overhead to not have to worry about pinning.

As will be discussed in the NUMA section, you can pin a process to both a CPU core and a NUMA node by running the following command (replacing the values for $phys_cpu_core and $associated_numa_node in addition to the game_server_executable name):

```
"numactl -physcpubind $phys_cpu_core -membind
$associated_numa_node ./game_server_executable"
```

Linux Scheduler

The default Linux scheduler is called the Completely Fair Scheduler (CFS),[21] and it is responsible for executing processes by taking care of the allocation of CPU resources. The primary goal of CFS is to maximize utilization of the vCPUs and, in turn, provide the best overall performance. If you don't pin game server processes to a vCPU then the Linux scheduler assigns threads for these processes.

There are a few parameters for tuning the Linux scheduler that can help with game servers. The primary goal of the three parameters documented below is to keep tasks on processors as long as reasonable given the activity of the task. We focus on the scheduler minimum granularity, the scheduler wakeup granularity, and the scheduler migration cost values.

To view the default value of all of the kernel.sched options run the following command:

```
sudo sysctl -A | grep -v "kernel.sched_domain" | grep
"kernel.sched"
```

The scheduler minimum granularity value configures the time a task is guaranteed to run on a CPU before being replaced by another task. By default

this is set to 3 ms on the C4 instance family when running Amazon Linux. This value can be increased to keep tasks on the processors longer. An option would be to double this setting this to 6 ms. Like all other performance recommendations in this whitepaper, these settings should be tested thoroughly with your game server. This and the other two scheduler commands do not persist the setting across reboots, so it needs to be done in a startup script:

```
sudo sysctl -w 'kernel.sched_min_granularity_ns=New_Value
```

The scheduler wakeup granularity value affects the ability of tasks being woken to replace the current task running. The lower the value the easier it will be for the task to force removal. By default this is set to 4 ms on the C4 instance family when running Amazon Linux. You have the option of halving this value to 2 ms and testing the result. Further reductions may also improve the performance of your game server.

```
sudo sysctl -w 'kernel.sched_wakeup_granularity_ns= New_Value'
```

The scheduler migration cost value sets the duration of time after a task's last execution where the task is still considered "cache hot" when the scheduler makes migration decisions. Tasks that are "cache hot" are less likely to be migrated, which helps reduce the possibility the task will be migrated. By default this is set to 4 ms on the C4 instance family when running Amazon Linux. You have the option to double this value to 8 ms and test.

```
sudo sysctril -w 'kernel.sched_migration_cost_ns= New_Value'
```


Memory

It is important that any customers running game servers on the c4.8xlarge instance pay close attention to the NUMA information.

Performance Tuning Option	Summary	Notes	Links or Commands
NUMA	On the c4.8xlarge NUMA can become	None of the C4 instance sizes	There are three options to deal with NUMA: CPU

Performance Tuning Option	Summary	Notes	Links or Commands
	an issue since there are two NUMA nodes.	smaller than the c4.8xlarge will have NUMA issues since they all have one NUMA node.	pinning, NUMA balancing, and the numad process.
Virtual Memory	A few virtual memory tweaks can provide a performance boost for some game servers.		Add the following to /etc/sysctl.conf: vm.swappiness = New_Value (Recommend start by halving the current value set for your system) Add the following to /etc/sysctl.conf: vm.dirty_ratio = New_Value (Recommend going with the default value of 20 on Amazon Linux) Add the following to /etc/sysctl.conf: vm.dirty_background_ratio = New_Value (Recommend going with the default value of 10 on Amazon Linux)

NUMA

All of the current generation EC2 instances support NUMA. NUMA is a memory architecture used in multiprocessing systems that allows threads to access both the local memory, memory local to other processors, or a shared memory platform. The key concern here is that the remote memory usage provides much slower access than the local memory. There is a performance penalty when a thread accesses remote memory, and there are issues with interconnect contention.

For an application that is not able to take advantage of NUMA, you want to ensure that the processor only uses the local memory as much as possible. This is only an issue for the c4.8xlarge instance because you have access to two processor sockets that each represent a separate NUMA node. NUMA is not a concern on the smaller instances in the C4 family since you are limited to a

single NUMA node. In addition, the NUMA topology will remain fixed for the lifetime of an instance.

The c4.8xlarge instance has two NUMA nodes. To view details on these nodes and the vCPUs that are associated with each node run the following command:

```
numactl --hardware
```

To view the NUMA policy settings run:

```
numactl --show
```

You can also view this information in the following directory (just look in each of the NUMA node directories):

```
/sys/devices/system/node
```

Use the numastat tool to view per-NUMA-node memory statistics for processes and the operating system. The –p option allows you to view this for a single process while the –v option provides more verbose data.

```
numastat -p process_name
numastat –v
```


CPU Pinning

There are three recommended options to address potential NUMA performance issues. The first is to use CPU pinning, the second is automatic NUMA balancing, and the last is to use numad. These options should be tested to determine which provides the best performance for your game server.

First we will look at CPU pinning. This involves binding the game server process both to a vCPU (or core) and to a NUMA node. You can use numactl to do this. Change the values for $phys_cpu_core and $associated_numa_node in addition to the game_server_executable name in the following command for

each game server running on the instance. See the numactl man page for additional options.[22]

```
numactl --physcpubind=$phys_cpu_core --
membind=$associated_numa_node game_server_executable
```

Automatic NUMA Balancing

The next option is to use automatic NUMA balancing. This feature attempts to keep the threads or processes in the processor socket where the memory that they are using is located. It also tries to move application data to the processor socket for the tasks accessing it. As of Amazon Linux Ami 2016.03, automatic NUMA balancing is disabled by default.[23]

To check if automatic NUMA balancing is enabled on your instance run the following command:

```
cat /proc/sys/kernel/numa_balancing
```

To permanently enable or disable NUMA balancing, set the `Value` parameter to 0 to disable or 1 to enable and run the following command:

```
sudo sysctl -w 'kernel.numa_balancing=Value'
echo 'kernel.numa_balancing = Value' | sudo tee
/etc/sysctl.d/50-numa-balancing.conf
```

Again these instructions are for Amazon Linux. Some distributions may set this in the /etc/sysctl.conf file.

Numad

Numad is the final option to look at. Numad is a daemon that monitors the NUMA topology and works to keep processes on the NUMA node for the core. It is able to adjust to changes in the system conditions. The article Mysteries of NUMA Memory Management Revealed explains the performance differences between automatic NUMA balancing and numad.[24]

To use numad you need to disable automatic NUMA balancing first. To install numad on Amazon Linux, visit the Fedora numad site and then download the most recent stable commit.[25] From the numad directory run the following commands to install numad:

```
sudo yum groupinstall "Development Tools"
wget https://git.fedorahosted.org/cgit/numad.git/snapshot/numad-
0.5.tar.gz
tar –zxvf numad-0.5.tar.gz
cd numad-0.5
make
sudo make install
```

The logs for numad can be found in /var/log/numad.log and there is a configuration file in /etc/numad.conf.

There are a number of ways to run numad. The numad –u option sets the maximum usage percentage of a node. The default is 85%. The recommended setting covered in the Mysteries of NUMA article is –u100, so this setting would configure the maximum to 100%. This forces processes to stay on the local NUMA node up to 100% of their memory requirement.

```
sudo numad –u100
```

Numad can be terminated by using the following command:

```
sudo /usr/bin/numad –i0
```

Finally, disabling NUMA completely is not a good choice because you will still have the problem with remote memory access so it is better to work with the NUMA topology. For the c4.8xlarge instance we recommend taking some action for most game servers. We recommend testing the available options that we discussed to determine which provides the best performance. While none of these options may eliminate memory calls to the remote NUMA node for a process, they each should provide a better experience for your game server.

You can test how well an option is doing by running your game servers on the instance and using the following command to see if there are any numa_foreign (i.e., memory allocated to the other NUMA node but meant for this node) and numa_miss (i.e., memory allocated to this node but meant for the other NUMA node) entries:

```
numastat -v
```

A more general way to test for NUMA issues is to use a tool like stress and then run numastat to see if there are foreign/miss entries:

```
stress --vm-bytes $(awk '/MemFree/{printf "%d\n", $2 * 0.097;}'
< /proc/meminfo)k --vm-keep -m 10
```

Virtual Memory

There are also a few virtual memory tweaks that we see customers use that may provide a performance boost. Again, these should be tested thoroughly to determine if they improve the performance of your game.

VM Swappiness

VM Swappiness controls how the system favors anonymous memory or the page cache. Low values reduce the occurrence of swapping processes out of memory, which can decrease latency but reduce I/O performance. Possible values are 0 to 100. The default value on Amazon Linux is 60. The recommendation is to start by halving that value and then testing. Further reductions in the value may also help your game server performance.

To view the current value run the following command:

```
cat /proc/sys/vm/swappiness
```

To configure this parameter to persist across reboots add the following with the new value to the /etc/sysctl.conf file:

```
vm.swappiness = New_Value
```

VM Dirty Ratio

VM Dirty Ratio forces a process to block and write out dirty pages to disk when a certain percentage of the system memory becomes dirty. The possible values are 0 to 100. The default on Amazon Linux is 20 and is the recommended value.

To view the current value run the following command:

```
cat /proc/sys/vm/dirty_ratio
```

To configure this parameter to persist across reboots add the following with the new value to the /etc/sysctl.conf file:

```
vm.dirty_ratio = New_Value
```

VM Dirty Background Ratio

VM Dirty Background Ratio forces the system to start writing data to disk when a certain percentage of the system memory becomes dirty. Possible values are 0 to 100. The default value on Amazon Linux is 10 and is the recommended value.

To view the current value run the following command:

```
cat /proc/sys/vm/dirty_background_ratio
```

To configure this parameter to persist across reboots add the following with the recommended value to the /etc/sysctl.conf file:

```
dirty_background_ratio=10
```

Disk

Performance tuning for disk is the least critical because disk is rarely a bottleneck for multiplayer game servers. We have not seen customers experience any disk performance issues on the C4 instance family. The C4 instance family only uses Amazon Elastic Block Store (EBS) for storage with no local instance storage; so C4 instances are EBS-optimized by default.[26] Amazon EBS can provide up to 48,000 IOPS if needed. You can take standard disk performance steps such as using a separate boot and OS/game EBS volume.

Performance Tuning Option	Summary	Notes	Links or Commands
EBS Performance	C4 instances are EBS-optimized by default. IOPS can be configured to fit the requirements of the game server.		NA

Benchmarking and Testing

Benchmarking

There are many ways to benchmark Linux. One option you may find useful is the Phoronix Test Suite.[27] This open source Python-based suite provides a large number of benchmarking (and testing) options. You can run tests against existing benchmarks to compare results after successive tests. You can upload the results to OpenBenchmarking.org for online viewing and comparison.[28]

There are many benchmarks available and most can be found on the OpenBenchmarking.org tests site.[29] Some tests that can be useful for benchmarking in preparation for a game server are the cpu,[30] multicore,[31] processor,[32] and universe tests.[33] These tests usually involve multiple subtests. Be aware that some of the subtests available may not be available for download or may not run properly.

To get started you need to install the prerequisites first:

```
sudo yum groupinstall "Development Tools" -y
sudo yum install php-cli php-xml -y
```

```
sudo yum install {libaio,pcre,popt}-devel glibc-{devel,static} -
y
```

Next download and install Phoronix:

```
wget https://github.com/phoronix-test-suite/phoronix-test-
suite/archive/master.zip
unzip master.zip
cd phoronix-test-suite-master
./install-sh ~/directory-of-your-choice/phoronix-tester
```

To install a test, run the following from the bin subdirectory of the directory you specified when you ran the install-sh command:

```
phoronix-test-suite install <test or suite name>
```

To install and run a test use:

```
phoronix-test-suite benchmark <test or suite name>
```

You can choose to have the results uploaded to Openbenchmark.org. This option will be displayed at the beginning of the test. If you choose "yes" you can name the test run. At the end a URL will be provided to view all the test results. Once the results are uploaded, you can rerun a benchmark using the benchmark result number of previous tests so the results are displayed side-by-side with previous results. You can repeat this process to display the results of many tests together. Usually you would want to make small changes and the rerun the benchmark. You can also choose not to upload the test results and instead view them in the command line output.

```
phoronix-test-suite benchmark TEST-RESULT-NUMBER
```

The screenshot below shows an example of the output displayed on OpenBenchmarking.org for a set of multicore benchmark tests run on the c4.8xlarge instance:

multicoretest

ptsh	test1	test2	test3	test4	test5	test6	test7
hmmer: Pfam Database Search	8.06	7.99	7.98	8.01	8.23	8.22	8.20
mafft: Multiple Sequence Alignment	4.08	3.98	4.25	4.23	4.27	3.88	3.85
gcrypt: CAMELLIA256-ECB Cipher	2250	2253	2247	2260	2260	2253	2257
john-the-ripper: Test: Blowfish	11404	11404	11404	11404	11411	7289	7300
x264: H.264 Video Encoding	295.01	297.65	295.59	296.63	286.41	249.39	249.40
himeno: Poisson Pressure Solver	1640.80	1646.83	1642.67	1638.73	1633.64	1640.49	1636.61
compress-7zip: Compress Speed Test	37151	37226	37259	37356	36301	28343	28418
build-apache: Time To Compile	24.84	24.75	24.76	24.82	24.85	25.61	25.53
build-php: Time To Compile	17.75	17.71	17.65	17.74	17.88	20.35	20.27
c-ray: Total Time	13.26	13.27	13.28	13.27	13.26	13.93	13.93
compress-pbzip2: 256MB File Compression	6.21	5.64	5.71	5.62	6.29	7.27	7.24
smallpt: Global Illumination Renderer; 100 Samples	48	47	48	48	48	62	62
bullet: Test: 1000 Convex	6.44	6.43	6.45	6.45	6.46	6.46	6.44
crafty: Elapsed Time	81.23	81.14	81.26	81.55	81.38	81.43	81.41
minion: Benchmark: Solitaire	95.28	93.71	95.50	95.25	95.58	94.68	94.10
minion: Benchmark: Quasigroup	156.10	154.39	156.19	155.48	156.41	154.58	154.68
povray: Total Time	75.58	75.52	75.76	75.99	75.64	90.61	90.58
openssl: RSA 4096-bit Performance	1067.27	1068.37	1067.83	1068.23	1068.10	973.00	973.13

OpenBenchmarking.org

CPU Performance Analysis

One of the best tools for CPU performance analysis or profiling is the Linux perf command.[34] Using this command you can record and then analyze performance data using perf record and perf report, respectively. Performance analysis is beyond the scope of this whitepaper, but a couple of great resources are the kernel.org wiki and Brendan Gregg's perf resources.[35] The next section describes how to produce flame graphs using perf to analyze CPU usage.

Visual CPU Profiling

A common issue that comes up during game server testing is that while multiple game servers are running (often unpinned to vCPUs) one vCPU will hit near 100% utilization while the other vCPUs will show low utilization. Troubleshooting this type of performance problem and other similar CPU issues can be a complex and time-consuming process. The process basically involves looking at the function running on the CPUs and finding the code paths that are the most CPU heavy. Brendan Gregg's flame graphs allow you to visually analyze and troubleshoot potential CPU performance issues.[36] Flame graphs

allow you to quickly and easily identify the functions used most frequently during the window visualized.

There are multiple types of flame graphs, including graphs for memory leaks, but we will focus on CPU flame graphs.[37] We will use the perf command to generate the underlying data and then the flame graphs to create the visualization.

First, install the prerequisites:

```
# Install Perf
sudo yum install perf

# Remove the need to use root for running perf record
sudo sh -c 'echo 0 >/proc/sys/kernel/perf_event_paranoid'

# Download Flamegraph
wget
https://github.com/brendangregg/FlameGraph/archive/master.zip

# Finally you need to unzip the file that was downloaded. This
will create a directory called FlameGraph-master where the flame
graph executables are located
unzip master.zip
```


To see interesting data in the flame graph you either need to run your game server or a CPU stress tool. Once that is running you run a perf profile recording. You can run the perf record against all vCPUs, against specific vCPUs, or against particular PIDs. Here is a table of the various options:

Option	Notes
-F	Frequency for the perf record. 99 Hz is usually sufficient for most use cases.
-g --	Used to capture stack traces (as opposed to on CPU function or instructions).
-C	Used to specify the vCPUs to trace.
-a	Used to specify that all vCPUs should be traced.
sleep	Specified the number of seconds for the perf record to run.

The following are the common commands for running a perf record for a flame graph depending on whether you are looking at all the vCPUs or just one. Run these commands from the FlameGraph-master directory:

```
# Run perf record on all vCPUs
perf record -F 99 -a -g -- sleep 60

# Run perf record on specific vCPUs specified by number after
the –C option.
perf record -F 99 -C CPU_NUMBER -g -- sleep 60
```

When the perf record is complete, run the following commands to produce the flame graph:

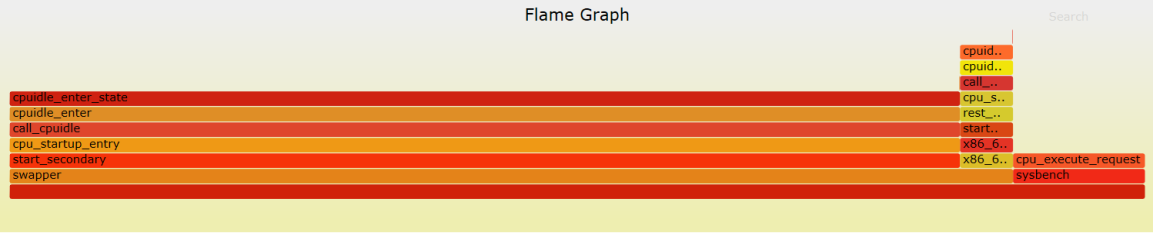
```
# Create perf file. When you run this you will get an error
about "no symbols found". This can be ignored since we are
generating this for flame graphs.
perf script > out.perf


# Use the stackcollapse program to fold stack samples into
single lines.
./stackcollapse-perf.pl out.perf > out.folded

# Use flamegraph.pl to render a SVG.
./flamegraph.pl out.folded > kernel.svg
```

Finally, use a tool like WinSCP to copy the SVG file to your desktop so you can view it.

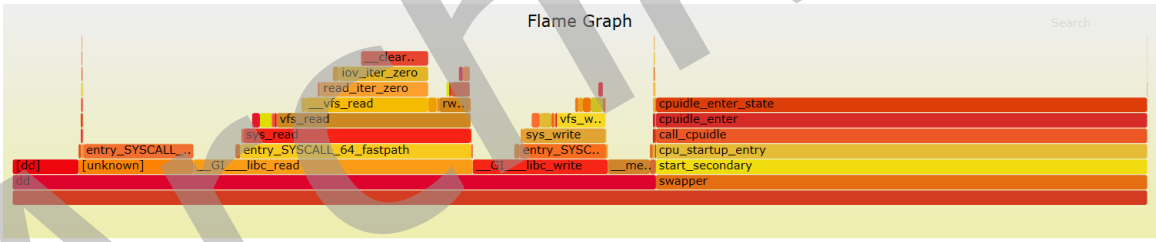
Below are two examples of flame graphs. The first was produced on a c4.8xlarge instance for 60 seconds while sysbench was running using the following options (for each in 1 2 4 8 16; do sysbench --test=cpu --cpu-max-prime=20000 --num-threads=$each run; done). You can see how little of the total CPU processing on the instance was actually devoted to sysbench. You can hover over various elements of the flame graphs to get additional details about the number of samples and percentage spent for each area.



The second graph was produced on the same c4.8xlarge instance for 60 seconds while running the following script:

```
(fulload() { dd if=/dev/zero of=/dev/null |dd if=/dev/zero
of=/dev/null |dd if=/dev/zero of=/dev/null |dd if=/dev/zero
of=/dev/null |dd if=/dev/zero of=/dev/null | dd if=/dev/zero
of=/dev/null | dd if=/dev/zero of=/dev/null | dd if=/dev/zero
of=/dev/null | dd if=/dev/zero of=/dev/null | dd if=/dev/zero
of=/dev/null | dd if=/dev/zero of=/dev/null | dd if=/dev/zero
of=/dev/null  & }; fulload; read; killall dd)
```

The output presents a more interesting set of actions taking place under the hood:



Conclusion

The purpose of this whitepaper is to show you how to tune your EC2 instances to optimally run game servers on AWS. It focuses on performance optimization of the network, CPU, and memory on the C4 instance family when running game servers on Linux. Disk performance is a smaller concern because disk is rarely a bottleneck when it comes to running game servers.

This whitepaper is meant to be a central compendium of information on the compute instances to help you run your game servers on AWS. We hope this guide saves you a lot of time by calling out key information, performance

recommendations, and caveats to get up and running quickly using AWS in order to make your game launch as successful as possible.

Contributors

The following individuals and organizations contributed to this document:

- Greg McConnel, Solutions Architect, Amazon Web Services
- Todd Scott, Solutions Architect, Amazon Web Services
- Dhruv Thukral, Solutions Architect, Amazon Web Services

Notes

[1] https://aws.amazon.com/ec2/

[2] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/c4-instances.html

[3] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

[4] https://aws.amazon.com/vpc/

[5] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[6] https://aws.amazon.com/about-aws/global-infrastructure/

[7] https://aws.amazon.com/ec2/faqs/#Enhanced_Networking

[8] https://en.wikipedia.org/wiki/Single-root_input/output_virtualization

[9] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[10] http://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/enhanced-networking-windows.html

[11] https://downloadcenter.intel.com/download/18700/Network-Adapter-Virtual-Function-Driver-for-10-Gigabit-Network-Connections

[12] https://www.kernel.org/doc/Documentation/networking/scaling.txt

[13] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html

[14] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html

[15] https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2/

[16] https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-windows-ec2/

[17] https://www.kernel.org/doc/Documentation/timers/timekeeping.txt

[18] https://xenbits.xen.org/docs/4.3-testing/misc/tscmode.txt

[19] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html

[20] https://raw.githubusercontent.com/brendangregg/msr-cloud-tools/master/showboost

[21] https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

[22] http://linux.die.net/man/8/numactl

[23] https://aws.amazon.com/amazon-linux-ami/2016.03-release-notes/

[24] http://rhelblog.redhat.com/2015/01/12/mysteries-of-numa-memory-management-revealed/#more-599

[25] https://git.fedorahosted.org/git/numad.git

[26] https://aws.amazon.com/ebs/

[27] http://www.phoronix-test-suite.com/

[28] http://openbenchmarking.org/

[29] http://openbenchmarking.org/tests/pts

[30] http://openbenchmarking.org/suite/pts/cpu

[31] http://openbenchmarking.org/suite/pts/multicore

[32] http://openbenchmarking.org/suite/pts/processor

[33] http://openbenchmarking.org/suite/pts/universe

[34] https://perf.wiki.kernel.org/index.php/Main_Page

[35] http://www.brendangregg.com/perf.html

[36] http://www.brendangregg.com/flamegraphs.html

[37] http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html