Evaluating Implications of Virtual Worlds on Server Architecture using Second Life

Srihari Makineni, Omesh Tickoo, Aaron Terrell, Jessica Young, Donald Newell
{srihari.makineni, omesh.tickoo}@intel.com, {enus}@lindenlab.com, {jessica.young, donald.newell}@intel.com

Abstract

Linden Lab's Second Life is the prominent Virtual World platform in the market today. Virtual Worlds like Second Life are emerging to be a main stream server workload because of their popularity due to richness of 3D content and immersive social experience they can provide. So, it is very important for computer architects to fully understand this workload and its requirements. In this paper, our goal is to fully analyze the performance and to characterize the processing of Second Life server Simulator process. The simulator process has three key critical functions that dominate the performance characteristics of this workload. These are: 1) Physics engine that is responsible for simulating real world behaviors taking into account mass of the objects, gravity, wind force, etc., 2) Scripting engine that is responsible for executing scripts attached to the objects. Scripts is the main way of manipulating object behaviors (motion, color, etc.) in-world on the server, and 3) Simulator logic that is responsible for simulating the world which includes avatar movement, calculating visible areas and communicating with the clients. Our work includes performance scaling experiments, comparison of performance on Intel's Clovertown and Nehalem processor based server systems and collecting and analyzing architectural characterization data for this workload. Our measurements have shown that Intel's latest Xeon servers using Nehalem processors offer 20 to 50% performance improvement over previous generation processor based system, and that the physics computation is more compute and memory intensive. To get a better perspective of Second Life's requirements, we have compared this workload with three other popular commercial server workloads (TPC-E, SPECjAppServer and SPECjbb) and found out that this workload executes 2 to 10 times more floating point, multiply and divide instructions.

1. Introduction

Virtual Worlds [19,20] represent a new emerging class of applications in the computer industry. These are gaining in popularity and growing in size rapidly in recent years because of rich and immersive user experience they provide with 3D graphics and real time interactions. Virtual worlds offer computer simulated worlds that are persistent. These virtual worlds, sometimes referred to as Meta Verse [21], and are being used for gaming, entertainment, socialization, education, training and collaboration. For example, US army uses virtual worlds to simulate different battle zone scenarios which are used to better train personnel in combat, rescue and recovery missions. Another example is that several education institutions have created presence in virtual worlds to increase their reach and enhance experience of distance learning. Users in virtual worlds feel a sense of presence through their avatars (online appearance) and can control these avatars to communicate, express emotions, move, dance and even to fly around in the virtual worlds.

Virtual worlds can be broadly characterized into two types. First type is Massively Multiplayer Online Games (MMOG [22] or MMORPG) such as World of Warcrafts [10], Eve Online [9], Club Penguin [11], LegoUniverse [12], etc, where users focus on role playing and playing games. In this type of virtual worlds, game developers envision and create games and define rules and objectives of games. The simulated world is mostly built by game developers and interactions and effects are predetermined. From computer processing perspective, the division of work between client and server is predetermined and most of the simulated world is downloaded or available to the client at the start of the game. As a result, the network bandwidth requirements are low and game developers take utmost care to minimize the network latency. These MMOGs adopt a technique, called sharding to support a large number of simultaneous users. They replicate the entire world on multiple sets of servers.

The second type of virtual worlds is where user's focus is not game playing, but more on interaction, socialization and commerce. In this type, users are allowed to build their own worlds as they imagined. Some examples of this type of virtual worlds are: Second Life [2], Olive [13], Quake [14], etc. Of these types of virtual worlds, Second Life from Linden Lab [15] is the most popular virtual world out there with more than 2000 multi-core servers simulating a world that spans tens of thousands of acres in size. In first quarter of 2009 alone, Second Life has registered about 124Million hours of user activity and in-world trading

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

amounting to about 120Million Dollars. Second Life has more than 75K users accessing different parts of the world at any given time. In spite of its popularity, there is very little understanding in the computer architecture community about its functionality and processing characteristics and requirements. It is important to understand these workloads now so we can build platforms that better support evolution of these workloads. There have been some published studies that have attempted to characterize the Second Life client [1, 23] and networking aspects [24] of Second Life, but there have been no published studies on Second Life server characterization. The main reason for this is lack of access to the Second Life server software. Linden Lab has recently started a pilot program to roll out a standalone version of their Second Life server software for private enterprise use. We have access to this standalone version of Second Life.

Our main contributions in this paper are: 1) Identify key frequently executed functional components of the Second Life server that dictate the overall performance and user experience, 2) Understand how these components perform on latest state of the art server systems, and their performance scaling characteristics, and 3) perform a detailed architectural characterization of SL processing and compare it to three other popular commercial server workloads (TPC-E [16], SPECjAppServer [18], SPECjbb [17]) to better understand this workload's requirements.

Rest of the paper is organized as follows. In section 2, we provide a quick high level overview of Second Life server side functionality and architecture. In section 3, we explain the test methodology, three Second Life scenarios we have created to understand and characterize the workload's key functional components. In section 4, 5 and 6, we go over the performance scaling, and characterization data. We also discuss how the Intel's latest Nehalem processor improves Second Life performance. Finally, in section 7, we provide our conclusions and discuss our future work in this area.

2. Second Life

In this section, we provide a brief overview of the Second Life (SL) server architecture and describe various functional components that make up the server side. Our intention here is not to provide a detailed overview. For detailed information, readers are requested to read the citations provided below.

There are two different flavors of SL server in existence today. One is a distributed version where different services are run on one or more servers. This is the public domain version of SL Grid that is hosted and managed by the Linden Lab. The second one is a grid in a box variant of SL where all the services run on a single server box. This version is developed for enterprises to use on enterprise networks behind firewalls. This is relatively a new offering from Linden Lab. Corporations and universities are using this version for collaboration and training purposes. This grid in a box version uses the same code base as the public domain version and any studies done on this are directly applicable to the public domain version as well. Figure 2.1 shows snapshots from some regions in SL which highlight the rich 3D content that its users have created.

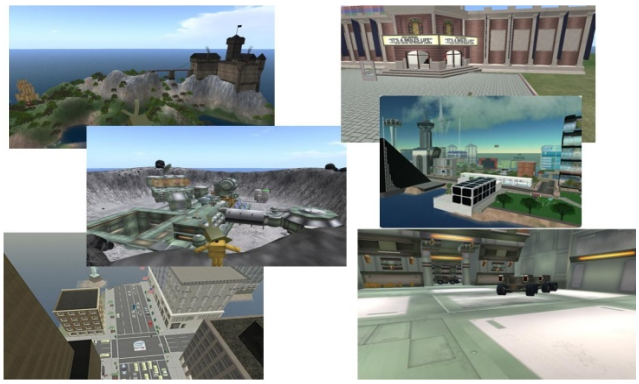

Figure 2.1. Snapshots of some Second Life regions

Users of SL buy land, create a terrain of their choice and build structures on it. The terrain typically includes mountains, water and some flat land. In SL, objects are built using simple basic shapes called primitives or prims. SL supports 8 different shapes of prims (cylinder, cube, sphere, etc) and 7 prim material types (wood, metal, concrete, etc). Users can change the basic prims by changing size, color, rotation, cut, shear, lighting and other attributes.

SL server consists of several services: Simulator, Map, Space, Asset, etc. There is also a backend mysql [25] database that acts as data provider and storage server for various services. In addition, Linden Lab's SL has its own currency, called Linden Dollar, and allows buying/selling of properties and other types of trading activity. SL server side architecture looks is shown in figure 2.2. We describe the main components next.

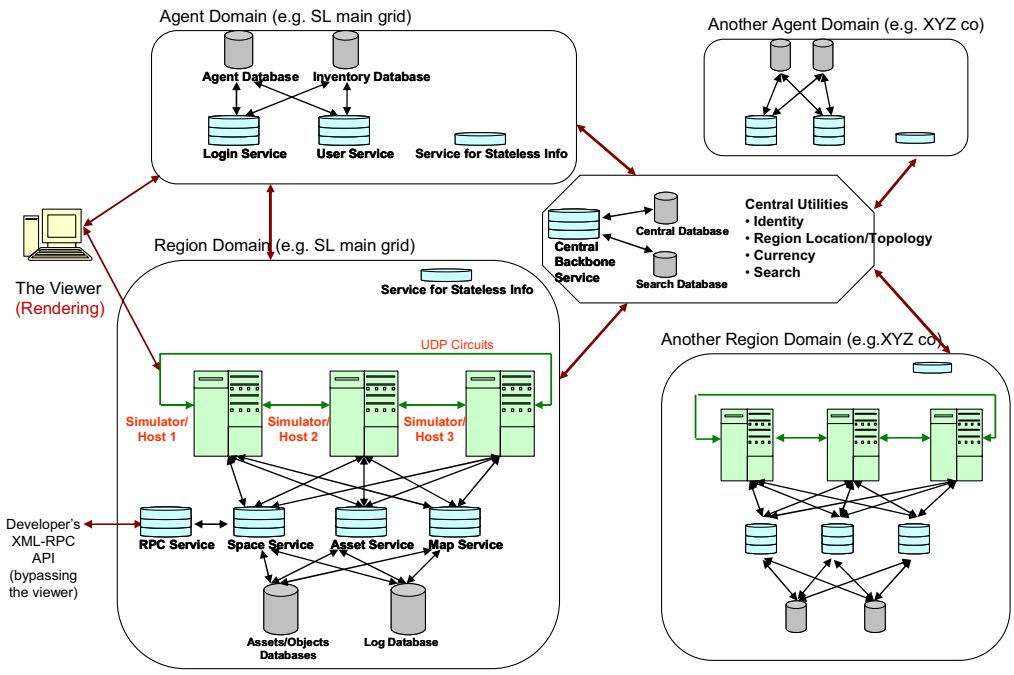

Figure 2.2. Second Life Server Architecture

Simulator

The main service that is responsible for simulating the world is the simulator. Each simulator is responsible for a 256m^2 region in the virtual world. This service (process) is responsible for storing object state, terrain information and performing visibility calculations to send to each connected viewer. This is the workhorse service of SL server. The simulator process talks to other services and the database server whenever is needed. The simulator process relies on TCP/IP and UDP/IP protocols for communicating with the clients. It uses TCP/IP internally for communicating with other services. The simulator process is currently single threaded and has a hard limit of 100 avatars per region. It also has an upper threshold of 45 frames/sec and won't process any more even if the CPU is not 100% utilized. Simulator processes communicate with each other, especially when avatars cross region boundaries. Within the simulator process, there are three main components. They are: Physics, Scripting and logic to handle avatars and compute visible area. We focus on these main components in this paper, and explain these in detail in the next section.

Map Server

This service is responsible for keeping track of information about the entire world being simulated. For example, when the user clicks on map button on the viewer, map server sends back map information that shows what regions are active, how many avatars are in each region, etc.

Asset Server

Every object inside SL is treated as an Asset and is assigned a unique ID (uuid). Asset server is responsible for managing this asset information. This also includes managing each avatar's inventory of objects (clothes, shoes, any objects built or bought). When the user clicks on the inventory button on the viewer, a request goes to the asset server and the asset server sends back asset information for that avatar to the client.

Space Server

This service is responsible for keeping track of which regions are where on the grid and maintaining neighbor information. It also handles routing of messages based on grid (x and y) locations. The simulator talks to the space server to register the region and to find out who the neighbors are.

Data Server

This service is responsible for handling connections to all the active databases (log, inventory, central, etc.). This handles communication with the database server on behalf of other services.

There are some other services to handle messaging, login, etc which we don't cover here due to space constraints.

3. Testing Methodology

In this section, we explain our testing methodology and describe the tools and systems used for the testing. First, we start with the server platforms used for this testing.

SERVER PLATFORMS

Figures 3.1 and 3.2 show details of Clovertown and Nehalem processor based server systems used for the studies. Even though these platforms support 2 sockets, we have used only one socket (one CPU) for our studies and

left the second one idle. We affinitized the simulator processes of interest to logical processors of one socket using taskaffinity tool available in schedutils [26] package for Linux OS. Nehalem system supports Hyper Threading (HT) which means each core has two logical processors (hardware threads) that share the core resources. HT allows for maximum utilization of the core resources. Nehalem also has a 256KB L2 cache that is private to each core. All the 4 cores of Nehalem share an 8MB L3 cache, while in Clovertown system there are two 4MB L2 caches per CPU each of which is shared by two cores. Nehalem also introduces a new system interconnect, called Quick Path Interconnect (QPI) which is link based instead of bus based like in the Clovertown system. QPI offers 12.8GB/s of read/write bandwidth per direction per link while the bus interconnect offers a total of 10GB/s. In Nehalem, the memory controller is integrated into the processor offering much lower latency to the memory.

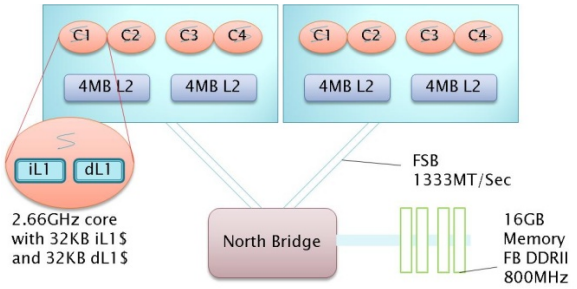

Figure 3.1. Details of Clovertown processor based Platform

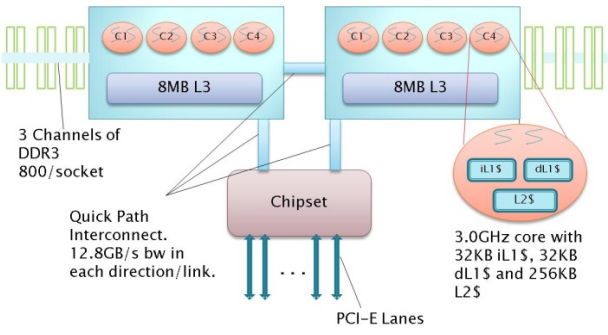

Figure 3.2. Details of Nehalem processor based Platform

We have enabled HT for some tests and disabled for some other tests. When we discuss individual test results, we point out whether HT is used or not.

We have turned off the HW prefetchers on both the platforms as we found that the HW prefetchers hurt performance slightly in some studies by increasing the cache miss rate.

TEST SETUP

Our test setup includes the server systems described above and several client machines. All the client machines are Nehalem (Core i7) processor based high end desktop systems with Gigabit NW interface and 4GB of memory. We made sure that the client machines are not a bottleneck in our studies. We run the grid in a box version of SL server on our server systems and run LLQABot script (described below) on the client machines. Servers and clients are connected to a Gigabit switch as shown in figure 3.3. Our server system is running Debian Linux version 4.0r8.

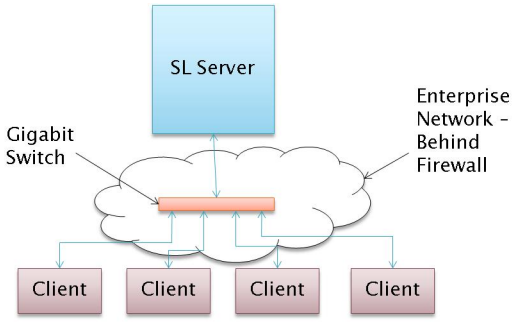

Figure 3.3. Test Setup

CLIENT SIDE PROGRAM

We use a publicly available bot generation and control tool to automate the process of logging in avatars into a region of SL and to control their actions. We call this program LLQABot [4]. Some enhancements were made to this tool to support wandering of avatars on the ground and while flying for some duration. Without this program, we literally need a person behind each avatar to control the avatar. Even then it will be difficult to generate repeatable actions for consistent and reproducible results. The LLQABot program solves this problem. Using this program, we can make avatars move, jump, fly, wander, query the database for inventory, rez and derez objects into the world, etc. In our studies, we mainly use this program for logging in the avatars and making them wander around the region. This program does not have any user interface associated with it, so no rendering of scenes has to be done. This allows the program to support multiple avatars at the same time. LLQABot sends commands and position updates to the server on behalf of each avatar and receives and processes screen updates and texture downloads from the server.

SCENARIOS

As we have explained in the section on SL (SL), there are a lot of different functions in SL that are of interest, but we

focus mainly on 3 very important aspects of the SL simulator (sim) process, which are physics processing, script processing and Avatar handling. The sim process performance is majorly dependent upon these three functional components. Some of the other functional components of sim process are network processing, processing large amount of data from the database (XML), etc. These have been studied in other domains extensively and their characteristics are well understood.

We have created scenarios to study the processing behavior of these three components in isolation as well as combined. Below is a brief description of each of the scenarios.

Physics – This scenario focuses on measuring performance of physics engine and interactions between it and the sim process. SL uses Havok [8] physics engine. This scenario allows us to study performance scaling and processing characteristics of the physics engine in isolation. To study this, we create a number of spheres in each SL region and place them in a valley. Since these are placed in a small valley, the spheres touch each other on the ground. We then apply a rotational force (llApplyRotationalForce [5]) to each of the spheres. Applying this force requires the physics engine to compute gravitational force to be applied based on object's mass and rotational direction. When this force is applied, the spheres spin along one of the X, Y or Z directions, and this causes lot of collisions which the physics engine has to resolve. This collision resolution is compute intensive. To minimize scripting overhead on this test, we applied rotational force randomly once every 0 to 10secs. Figure 3.4 illustrates this scenario in one of the regions.

Figure 3.4. Physical Objects spinning and colliding

Scripting – This scenario focuses on measuring scripting engine's performance inside SL sim process. SL uses Mono runtime engine to host scripts. Mono[6] is an open source cross platform implementation of .NET[7] development framework. The scripting language supported by SL sim process is called Linden Scripting Language (LSL). For this scenario, we instantiate a number of 0.25m wide cubes and place them in stack of 10x10 grids. These cubes are not marked physical in the simulator, so they can hang in the air. These cubes don't touch each other and each cube has a script that runs as fast as the simulator process allows. Whenever this script runs, it tries to change color of the cube and rotate it along x or y axis. The simulator process has to register the timers when the object is instantiated and fire off the scripts every time the timer goes off. This scenario allows us to measure performance of this component and to characterize the processing. This scenario is illustrated in figure 3.5.

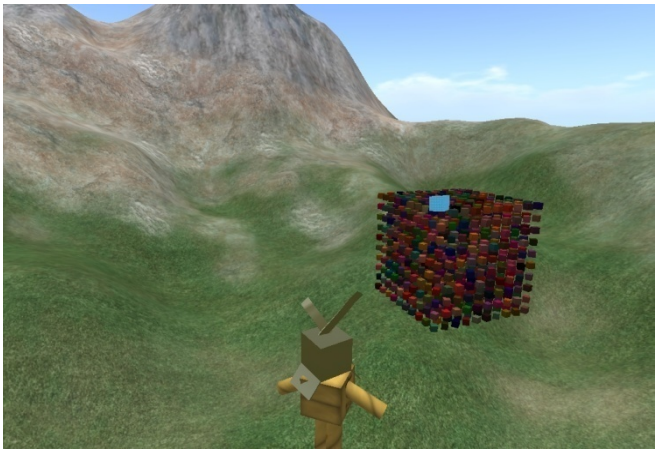

Figure 3.5. Scripted cubes rotating and changing color

Avatars – In this scenario, we focus on sim's ability to process avatars and their movement around the region. We place a number of avatars in each region and make the avatars move randomly around the region. We use LLQABot program on the client machines to log in and control these avatars. SL simulator does not know or care who is controlling the bots and treats each avatar same as if a person is logged into the region using a SL viewer. So, if 75 avatars logged in using LLQABot program, the load on the SL server would be the same as if 75 people logged in from different SL viewers. Whenever an avatar moves, the simulator has to update the avatar's new position with respect to other avatars and objects and send updates to all the other avatars within some visible range. Figure 3.6 is a snapshot of a region with some number of avatars (bots) moving around.

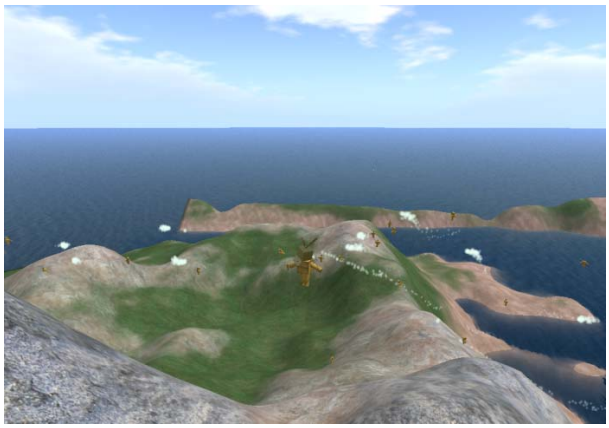

Figure 3.6. Avatars wandering on the ground and flying

Mixed – In this scenario, we combine all the above key elements of SL sim process and create a mixed scenario. This scenario is probably a better representative of real world load on SL servers. So, understanding the processing and performance of this scenario is quite important as well. This scenario has either 50 or 75 agents moving around, 800 non physical cubes rotating and changing color and 100 spinning spheres causing collisions.

4. Scaling Experiments

Before we jump into detailed characterization of SL performance, it is useful to first understand how the three main components of SL sim process perform in isolation and how their performance scales with varying amounts of load. To understand this, we look at CPU utilization and Time/Frame metrics. The Time/Frame metric is reported by the SL sim server. This indicates how many frames (scene updates) that the sim process is able to compute per sec. The SL sim process has an upper threshold of 45 frames/sec (22.2ms/frame). We start out with the Physics engine.

Figure 3.7. Avatars, spinning spheres and rotating cubes

PHYSICS

The graph in figure 4.1 shows how the SL sim process scales with increased number of physical objects, spheres in this case, spinning and colliding with each other. The primary y-axis in the graph shows CPU utilization and secondary y-axis shows time/fr numbers. Data shown in the graph is for 8 sims running on 8 hardware threads (4 cores with HT) of a Nehalem socket. Each sim process is loaded with a number of physical spheres which is shown on the x-axis. The graph shows that CPU utilization and time/fr doubling from 200 to 400 spheres and more than doubling from 400 to 600 spheres. The CPU utilization reaches 100% at 600 spheres. Increasing load from 600 to 800 spheres causes time/fr increase beyond the desired 22.2ms which means that the sim process is not able to keep up 45fr/sec frame rate. We will discuss the nature of this physics processing in the next section.

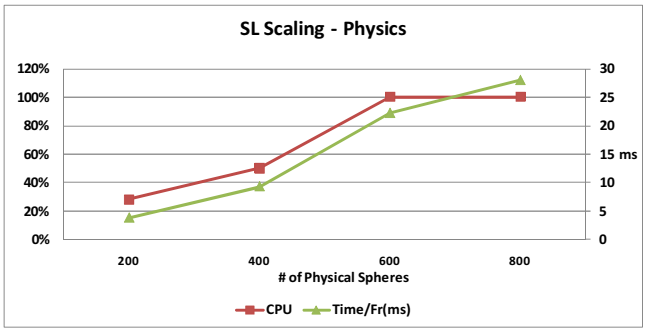

Figure 4.1. SL physics engine performance with Number of Physical Objects

SCRIPTING

Graph in figure 4.2 shows how SL sim process performance scales with number of scripted objects. We varied the number of scripted objects from 800 to 4800 (shown on x-axis). CPU utilization (primary y-axis) and time/fr (secondary y-axis) scale more linearly than the physics scenario until we reach 3200 scripted cubes. Just like in the Physics scaling experiment above, we have loaded 8 sims on 8 hw threads. Each sim is processing the same number of scripted objects. Surprisingly, both CPU and time/fr flatten out after 3200 scripted cubes. To understand what is happening, we looked at events executed per sec metric, also reported by the SL sim process. This data is shown in table 4.1. This data reveals that the sim process executes fewer events/sec beyond 3200 scripted objects indicating that the sim process throttles the events processed sec to maintain the 45 frames/sec frame rate.

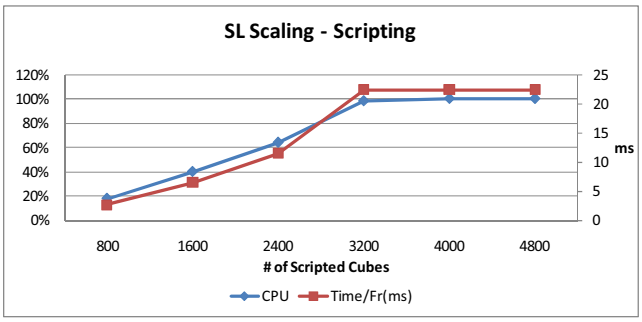

Figure 4.2. SL Performance Scaling with Number of Scripted Objects

# of Cubes	Events/sec
2400	10000
3200	6700
4000	6000
4800	4750

TABLE 4.1. Throttling of Events/sec with increased number of scripted objects

AVATARS

Now, we will look at how the SL sim process handles avatars (agents) and their movement. Just like the other two experiments, this one also starts 8 sims on 8 hw threads and each sim handles same number of avatars. After these avatars login, they start walking randomly around the region. The client program that logs in these avatars constantly sends avatar position updates to the server forcing the server to calculate screen updates. While running this test, the physics and scripting load on the SL sim process is very small and most of the activity is in the sim logic that handles the avatars. As we increase the number of avatars from 25 to 95 the CPU utilization increases somewhat linearly. Even at 95 avatars, CPU utilization is only 60% (all cores of the CPU are at 60%). The SL sim process has a built-in limitation of 100 avatars/region.

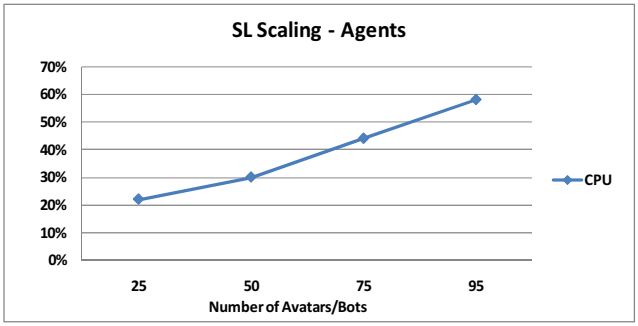

Figure 4.2. SL Performance Scaling with Number of Agents

We have seen how the main components of SL sim process respond to varying amounts of load, we will try to analyze and characterize this processing in more detail in the next section. We will also compare SL sim processing with 3 other popular commercial server workloads to get a better perspective at the requirements of SL server processing.

5. SL vs. Other Server Workloads

In this section, we compare SL behavior to other well known commercial server workloads. This kind of comparison helps architects get a better perspective about this workload's requirements. We compare SL to TPC-E, SPECjAppServer and SPECJbb workloads on various metrics. Data for all the workloads is collected on Intel Nehalem processor based dual socket server systems.

CYCLES PER INSTRUCTION (CPI) COMPARISON

CPI metric indicates, on average, how many clocks the CPU requires to retire one instruction. CPI value depends on several factors including frequency, micro-architecture, cache size, memory latency, etc. The graph in figure 5.1 shows how CPI numbers for various SL scenarios compare to other server workloads.

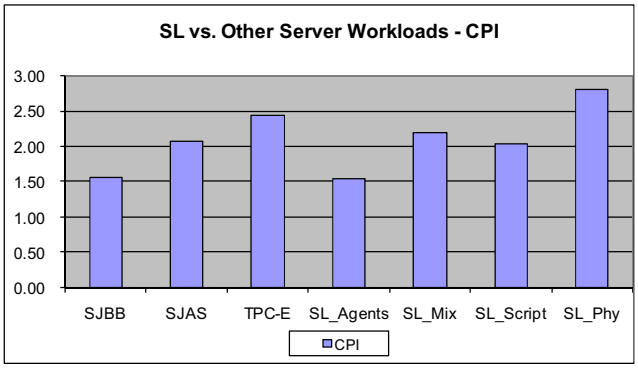

Figure 5.1. CPI Comparison

Except for SL_Phy scenario, all the other SL scenarios have lower CPI than TPC-E. This is expected because the physics engine is constantly trying to resolve collisions amongst 800 spinning spheres, which requires lot of state management and vector processing. High CPI of SL_Physics scenario is mainly due to two reasons. These are: 1) higher LLC MPI (explained in next sub section of the paper and shown in figure 5.3) is due to large amount of active memory that the physics engine allocates to keep track of 800 objects/sim, and 2) resolving collisions involves execution of complex instructions such as floating-point, multiply, divide and SSE instructions. On the other hand, SL_Agents has the lowest CPI. This is because the processing involved in handling the avatars and their movement is light weight compared to others scenarios. Looking at the CPI number (2.19) for SL_Mix scenario, which is a better representative of real world load on SL, we can say that SL in general is on the high side of

CPI spectrum. Next, we will look at some other metrics that can explain observed CPI behavior.

MISSES PER INSTRUCTION (MPI) COMPARISON

We will start with the L2 MPI. To remind readers, the Nehalem processor has a 256KB of L2 per core that is shared by two hardware threads. The graph in figure 5.2 shows L2 MPI numbers.

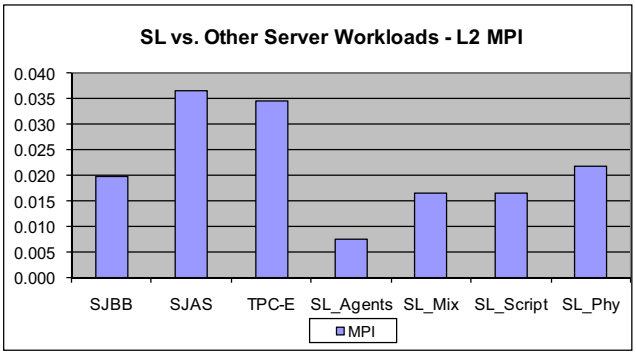

Figure 5.2. L2 MPI Comparison

Looking at the data in the graph, it is clear that SL in general has lower L2 MPI. SL_Mix has a L2 MPI of 0.016 which is less than that of SJBB. This shows that both the code and data of SL has high temporal locality. Even though the MPI of SL_Mix is lower, it has high CPI as shown in figure 5.1. This is because of the nature of instructions and not due to cache capacity. We will look at the instruction mix shortly, but before we do that let's look at the last level cache performance first. The graph in figure 5.3 shows last level cache MPI and memory read bandwidth numbers.

SL_Phy has highest LLC MPI of 0.011 while the SL_Agents has the least (0.0022). Physics engine allocates about 12 to 15MB of memory to manage state information for 800 spheres. This is per region and there are 8 regions active on the socket. So, the physics engine's memory footprint far exceeds the available cache (8MB) causing large number of misses. This is also evident from the miss ratio (misses/requests) numbers. Miss ratio for SL_Phy is 51% while it is only 36% for the SL_Mix scenario. Comparing SL_Mix with SJAS, we can see that SJAS has much higher L2 MPI, but lower LLC MPI. This is because the 8MB cache is able to accommodate SJAS code and data better. Plus, SJAS has a lot of sharing across its threads vs. no sharing at all across multiple SL regions.

Even though SL_Phy has highest MPI, it's memory read bandwidth is less than SJBB. This is because SJBB has much lower CPI than SL_Phy, hence is doing lot more work per sec (instructions/sec) and as a result has higher memory read bandwidth (misses/sec). However, the writeback bandwidth numbers for SL are generally lower than other workloads. Overall, SL's memory bandwidth requirements fall within the range of other three server workloads.

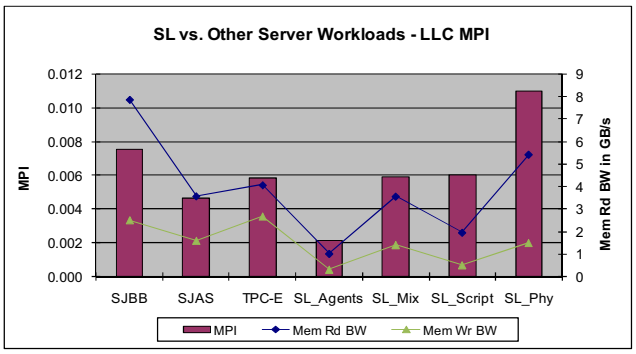

Figure 5.3. LLC MPI and Mem Read BW Comparison

INSTRUCTION MIX

Table 5.1 compares numbers for three different types of instructions executed by the workloads. In general, SL executes large number of floating, multiply and divide instructions which explains partly why it's CPI is higher. Another interesting point to note here is that SL_Phy_800 (800 spheres) has fewer of these instructions executed than SL_Phy_600 (600 spheres) because the cores are waiting for memory more often hence end up doing less amount of work per sec in case of SL_Phy_800 case.

It is also interesting to note that SL_Agents scenario executes a large number of floating point, multiply and divide operations, yet it's CPI is much lower than SL_Phy and SL_Mix scenarios. This is because SL_Agents has smaller cache footprint hence don't end up spending very much time waiting for data.

	TPC-E	SJBB	SJAS	SL_Agents	SL_Mix	SL_Scripts	SL_Phy_800	SL_Phy_600
FP Ops	1	5	8	20	11	4	4	13
Multiply Ops	16	84	48	231	225	150	330	354
Divide Ops	2	10	15	76	54	20	37	55

Table 5.1. Comparison of Instruction Mix across workloads

Looking at CPI, L2 and LLC MPI, memory bandwidth and instruction mix numbers across these four workloads, it is clear that the SL workload's requirements fall within the range of other three more established server workloads, except for the instruction mix. The SL seems to be executing a lot more of floating point, multiplication and

divide instructions. Also, physics intensive scenarios need larger caches to provide better performance.

6. SL Performance on Nehalem vs. CloverTown

In this section, we evaluate SL sim performance on two latest Intel server platforms, Clovertown and Nehalem processor based systems. The objective of this comparison is to figure out how much SL can benefit from the integrated memory controller, faster memory and improved cache hierarchy and micro architecture of the Nehalem processor. For this comparison, we load 4 sims on each system (no HT). Later in this section, we will also evaluate the impact of Hyperthreading that is available in Nehalem on SL's performance.

Graph in figure 6.1 plots efficiency of SL sim process on Nehalem over Clovertown. Efficiency is calculated using time/fr metric for 5 different scenarios shown on the x-axis. Data in the graph shows that Nehalem system provides 40 to 50% benefit over the Clovertown, except for the 1sim, 5000 cubes scenario. For this case, we have compared number of events/sec metric and found out that the Nehalem system is generating almost twice as many events (9900). To understand where the efficiencies are coming from, we have collected processor performance counters on both the platforms. Graph in Figure 6.2 shows CPI and MPI efficiencies for three different scenarios. Nehalem has 10 to 20% lower MPI and 25 to 45% lower CPI than the Clovertown processor based system. Even though both the processors have a total of 8MB cache per CPU, the Nehalem cache is more efficient, thanks to its improved replacement policy and organization. CPI improvement is not entirely due to improvement in MPI, but other improvements are also contributing. For example, for the physics scenario (800 spheres), we have compared memory latency and it is about 70ns for Nehalem and about 150ns for Clovertown.

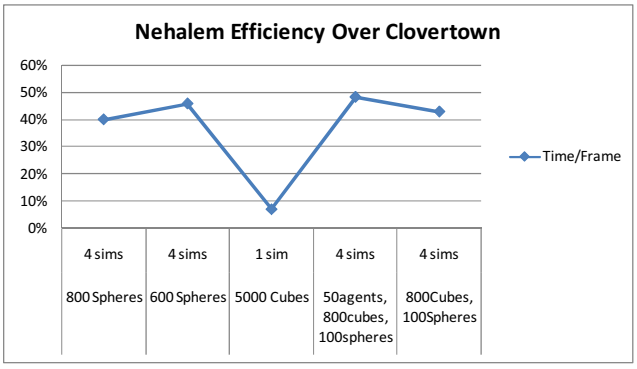

Figure 6.1. Nehalem benefit for SL – Time/Frame

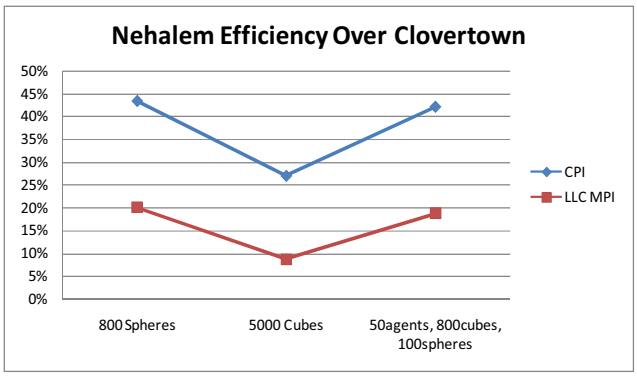

Figure 6.2. Nehalem benefit for SL – CPI and MPI

HYPER THREADING (HT) ADVANTAGE

We have seen that Nehalem processor without HT offers significant advantage over the previous generation Clovertown processor. Now, we turn our attention to HT and figure out if HT offers any additional benefit to SL sim process. We have to remind readers one more time that the SL sim process is not multi-threaded, and as such can only utilize one logical processor. So, the real advantage of HT for SL is to double the number of sims per box. We can run 8 sims per CPU (socket) with HT on vs. 4 when HT is off. But when HT is on, two hw threads share the core resources (execution ports, TLBs, caches, etc.). This can lead to lower performance on each logical processor. The following tables show impact of HT on SL sim process for two different scenarios. Table 6.1 shows impact for the mixed scenario where each sim is handling 50 avatars moving randomly around the region, 800 cubes rotating and changing color and 100 spinning spheres. Going from 4 sims to 8 sims per CPU slows down each sim significantly, so if sims are heavily loaded then using HT to double the number sims is not very beneficial. However, there is still some benefit with HT. Looking at the Fr/sec metric in Table 6.1, we can see that 8 sims delivers 43 fr/sec/sim (344) vs. 4 sims deliver 71 fr/sec/sim (284). This amounts to roughly 20% more frames/sec in 8 sim case. Frames/sec number is calculated based on time/fr, not measured directly (sim process has a upper threshold of 45 frames/sec). It must be noted that the CPI and MPI numbers didn't double going from 4 sims to 8 sims which again confirms that there is benefit to be had with HT.

	4 sims	8 sims
	4 cores	8 threads
CPU	87%	100%
Cycles/100Inst	149	219
L2 Req/10KI	239	366
L2 Miss/10kI	105	164
LLC Miss/10KI	43	59
Fr/Sec	71	43

Table 6.1. HT Impact – 50 Agents, 800 Rotating Cubes, 100 Spinning Spheres

Table 6.2 shows similar data but for a physics scenario with 600 spinning spheres per region. Here also CPI didn't double from 4 to 8 sims and Fr/sec numbers show that we get about 11% more frames/sec with HT.

	4 sims	8 sims
	4 cores	8 threads
CPU	72%	100%
Cycles/100Inst	164	281
L2 Req/10KI	297	495
L2 Miss/10kI	140	217
LLC Miss/10KI	55	110
Fr/Sec	81	45

Table 6.2. HT Impact – 600 Spinning Spheres

If the sim process were fully multi threaded or even if it has a way to offload one major component to a different thread then in that case HT would help scale up individual sim's performance. Having this support in the software provides flexibility and supports both scale up and scale out options.

7. Conclusions and Future Work

Virtual Worlds are gaining popularity and growing in size rapidly due to rich 3D content and immersive experience they provide. Yet, there have been no published studies on what this workload's requirements on server platforms are. We have done a thorough analysis of one of the most popular virtual world server, called Second Life. Our analysis included analyzing how the simulator process performance scales with varying amounts of physics, scripting and avatar load. This analysis showed that the physics engine is the most demanding of all in terms of compute and cache resources. We have also compared SL performance on Nehalem to Clovertown and showed that Nehalem processor based systems can improve SL performance by 20 to 40%. Also, HT on Nehalem doubles the number of regions per socket and still improves performance by about 5 to 20%. For cases, where performance scale up is more important, HT can help there too if the sim is multithreaded. We have collected detailed architectural characterization data for the sim process to understand its requirements and compared this data with similar data from three other well known server workloads (TPC-E, SPECjbb and SPECjAppServer). This comparison (CPI, MPI, MemBW and Mem Latency) revealed that SL sim process's requirements fall within the range of these well known server workloads. SL simulator process issues 2 to 10 times more floating point, multiply and divide instructions than the three other workloads.

Future work in this area includes understanding performance implications of sim multithreading options, creating more complex scenarios involving other services, performing cache and frequency scaling experiments, etc.

References

[1] Sanjeev Kumar, Jatin Chhugani, Changkyu Kim, Daehyun Kim, Anthony Nguyen, Pradeep Dubey, Christian Bienia, Youngmin Kim, "Second Life and the New Generation of Virtual Worlds," *Computer*, vol. 41, no. 9, pp. 46-53, Sept. 2008, doi:10.1109/MC.2008.398

[2] Second Life, http://www.secondlife.com.

[3] Nebraska, https://blogs.secondlife.com/community/workinginworld/blog/2009/04/01/second-life-lives-behind-a-firewall

[4] LLQABot, http://lib.openmetaverse.org/wiki/Main_Page

[5] LSL, http://en.wikipedia.org/wiki/Linden_Scripting_Language

[6] Mono, http://www.mono-project.com/Main_Page

[7] .NET, http://www.microsoft.com/net/

[8] HAVOK, http://software.intel.com/sites/havok/

[9] Eve Online, http://www.eveonline.com/

[10] World Of Warcraft, http://www.worldofwarcraft.com/index.xml

[11] Club Penguin, http://www.clubpenguin.com/

[12] Lego Universe, http://universe.lego.com/en-us/Default.aspx?domainredir=www.legouniverse.com

[13] Olive, http://www.forterrainc.com/

[14] Quake, http://www.quakelive.com/

[15] Linden Lab, http://lindenlab.com/

[16] TPC-E, http://www.tpc.org/tpce/tpc-e.asp

[17] SPECjbb, http://www.spec.org/jbb2005/

[18] SPECjAppServer, http://www.spec.org/jAppServer2004/

[19] Innovation in VirtualWorlds. http://www.research.ibm.com/virtualworlds/.

[20] Virtual World, http://en.wikipedia.org/wiki/Virtual_world

[21] Meta Verse, http://en.wikipedia.org/wiki/Metaverse

[22] MMOG, Massively Multiplayer Online Games, http://en.wikipedia.org/wiki/MMOG

[23] Second Life Viewer, Client program to connect to Second Life grip, http://secondlife.com/support/downloads.php

[24] Kinicki, J. and Claypool, M. 2008. Traffic analysis of avatars in Second Life. In *Proceedings of the 18th international Workshop on Network and Operating Systems Support For Digital Audio and Video* (Braunschweig, Germany, May 28 - 30, 2008). NOSSDAV '08. ACM, New York, NY, 69-74. DOI= http://doi.acm.org/10.1145/1496046.1496063

[25] MySQL, open source database server, http://www.mysql.com/

[26] SchedUtils for Linux, http://sourceforge.net/projects/schedutils