LIGO Grid Computing

INTERNATIONAL CENTRE for THEORETICAL SCIENCES

TATA INSTITUTE OF FUNDAMENTAL RESEARCH

Project Report

by

Samim Zahoor

IT 03/13

Acknowledgments

I wish to thank my parents for their immense support in everything I have ever attempted. They have always stood behind me in my endeavours. All that I am today is because of them.

I also wish to thank Professor P. Ajith for allowing me to work under him. Also, I wish to express my gratitude to all the people at ICTS TIFR, Bangalore, especially my guide Mr. Srinavas, for helping me when I most needed it.

Lest its side-effects be ignored, I also feel compelled to mention the countless cups of coffee that kept me going through dark nights.

Samim Zahoor,
NIT Srinagar

Contents

1 Introduction **5**

2 Hardware Configuration **5**
- 2.1 General Configuration 6
- 2.2 Choice of Server 7
- 2.3 Network and Storage 8
- 2.4 Operating System 9

3 My Work **9**
- 3.1 Introduction 9
- 3.2 HTCondor 9
  - 3.2.1 High-Throughput Computing (HTC) and its Requirements 9
  - 3.2.2 HTCondor's Power 10
  - 3.2.3 Installation and Configuration 12
  - 3.2.4 Security 15
- 3.3 User priorities and job policy 16

4 IO Benchmarking **17**

List of Figures

1 Graphical Representation of Organization 8
2 Write performance for ext4 18
3 Read performance for ext4 19
4 Write performance for ZFS RAIDz2 20
5 Read performance for ZFS RAIDz2 20
6 Write performance for ZFS RAIDz2 with cache 21
7 Read performance for ZFS RAIDz2 with cache 21

1 Introduction

The Astrophysical Relativity research group at ICTS-TIFR is part of the LIGO Scientific Collaboration – a group of some 950 scientists from 16 countries involved in the commissioning of the LIGO instruments and in analyzing the data to make discoveries. Gravitational-wave data analysis is computationally intensive, and is performed at large computing clusters located at different parts of the globe, which form the LIGO data grid. I joined as Project Intern at the computing facility that was being set up at ICTS-TIFR. It started as a Tier-3 grid computing center and will be upgraded to a Tier-2 center in the next two years.

The computing cluster at ICTS-TIFR features 25 teraflops of peak computing power and 200 terabytes of dedicated storage – one of the biggest of its kind in India. I had the opportunity to work with cutting-edge grid-computing technologies, high-performance file systems and storage solutions, high-throughput computing frameworks, federated identity solutions, etc., and to interact closely with the scientists working on big data.

2 Hardware Configuration

The facility was built as a Beowolf style computing cluster.Beowulf is a multi-computer architecture which can be used for parallel computations. It is a system which usually consists of one server node, and one or more client nodes connected via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running a Unix-like operating system, with standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the FreeBSD, Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb, the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a disk-less client configuration, a client node doesn't even know its IP address or name until the server tells it.

One of the main differences between Beowulf and a Cluster of Workstations (COW) is that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard.

Beowulf is not a special software package, new network topology, or the latest kernel hack. Beowulf is a technology of clustering computers to form a parallel, virtual supercomputer. Although there are many software packages such as kernel modifications, PVM and MPI libraries, and configuration tools which make the Beowulf architecture faster, easier to configure, and much more usable, one can build a Beowulf class machine using a standard Linux distribution without any additional software. If you have two networked computers which share at least the /home file system via NFS, and trust each other to execute remote shells (rsh), then it could be argued that you have a simple, two node Beowulf machine.

2.1 General Configuration

The configuration of the hardware components used to build the system is as follows.

- 2 head nodes (each with 2 processors, 256GB memory)
- 2 file servers or storage nodes (each with 2 processors, 256GB memory, 48 TB or higher usable storage)
- 16 slave nodes (each with 2 processors, 128GB memory, 1TB of local disk)
- 16 slave nodes (each with 2 processors, 64GB memory, 1TB of local disk)
- Gigabit ethernet switch with 48 Gb ports and 8 10Gb ports (for head nodes and storage nodes)

2.2 Choice of Server

A rack unit, U or RU as a unit of measure describes the height of electronic equipment designed to mount in a 19-inch rack or a 23-inch rack. The 19 inches (482.60 mm) or 23 inches (584.20 mm) dimension reflects the width of the equipment mounting-frame in the rack including the frame; the width of the equipment that can be mounted inside the rack is less. One rack unit is 1.75 inches (44.45 mm) high.

The height of rack-mounted equipment is frequently described as a number in "U". For example, one rack unit is often referred to as "1U", 2 rack units as "2U" and so on.

- The 1U form factor is typically used when 1 to 2 sockets are needed in a node PCI-e card(s) are needed when certain functionality is not on the motherboard (for example, IB card, PCI-e card to Nvidia Tesla 1U system) Density is not an issue Smaller clusters (blades are usually more expensive than 1U nodes for smaller clusters) Systems that need added or larger hard drives with hot-swap capability that blades can't provide (blades typically have a small number of drives that are small in form factor and capacity and may or may not be hot-swappable)

- 2U form factor nodes are typically used when 2 to 4 sockets (typically 2 sockets) are needed in a node More hard drives are needed than you can put in a 1U form factor There is a possibility of adding PCI-e cards that have functionality not on the motherboard (for example, new IB card, PCI-e card to Nvidia Tesla 1U system) Multiple network connections are needed (such as a master/login node) Redundant power supplies are needed (although, many 1U nodes have redundant power supplies) The nodes need to use less power than a 1U (a 2U node has larger fans than a 1U and typically uses less power)

- The 4U form factor nodes are typically used when a large number of sockets is needed in a single node (typically 4 sockets) A large number of DIMM slots (lots of memory) is needed in a single node A large number of hard drives are needed in the node A large number of PCI-e cards are needed with a single node The nodes need to use less power than a 2U (a 4U node has larger fans than a 2U and typically uses less power)

- Blades are typically used in situations when high density is critical Power and cooling is absolutely critical (blades typically have better power and cooling than rack-mount nodes) The applications don't require much local storage capacity PCI-e expansion cards are not required (typically blades have built-in IB or GbE) Larger clusters (blades can be cheaper than 1U nodes for larger systems)

With these general guidelines in mind, 4U solution with 8 nodes was used for the compute nodes. For servers 2U and 1U solutions used.

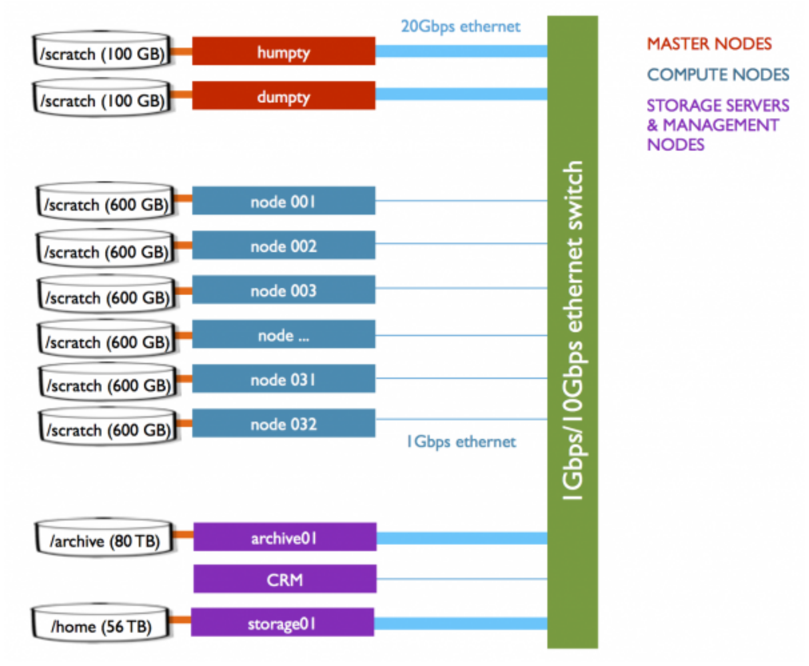

Figure 1: Graphical Representation of Organization

2.3 Network and Storage

A switch with 48 number of 1Gb ports and 8 number of 10Gb ports was used for newtorking. The 10Gb ports were reserved for File

Servers, CRM and the Submit Servers. Compute nodes used 1Gb ports for networking.

For the file servers, 48TB or more of usable space, 7200 RPM, RAID5 with hot spare. Hitachi Ultrastar or Western Digital or equivalent disk. For the /local on nodes, 1TB SATA disk with 10,000 RPM.

2.4 Operating System

For all the systems on the cluster we used the **Debian 7.9** because it is very lightweight and has good support for most of the software that was being used by the researchers of the LIGO group.

3 My Work

3.1 Introduction

Over the course of 3 months starting from to , I worked on setting up the job submission platform, setting up cluster monitoring services [**?**] is best seen as an amalgamation and extension of the aforementioned algorithms.

3.2 HTCondor

3.2.1 High-Throughput Computing (HTC) and its Requirements

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High- Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of FLoating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year. Their problems are of a much larger scale. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete. The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time. As computers became smaller, faster, and cheaper, users moved away from centralized mainframes and purchased personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. The personal computer is slower than the large centralized machine, but it provides exclusive access. Now, instead of one giant computer for a large institution, there may be hundreds or thousands of personal computers. This is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may rise dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping).

3.2.2 HTCondor's Power

HTCondor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. HTCondor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to HTCondor. HTCondor finds an available machine on the network and begins running the job on that machine. HTCondor has the capability to detect that a machine running a HTCondor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. HTCondor continues the job on the new machine from precisely where it left off. In those cases where HTCondor can checkpoint and migrate a job, HTCondor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains. HTCondor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to HTCondor. Depending upon the number of machines in the HTCondor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment. HTCondor does not require an account (login) on machines where it runs a job. HTCondor can do this because of its remote system call technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted. HTCondor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. HTCondor implements ClassAds, a clean design that simplifies the user's submission of jobs. ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the HTCon- dor pool advertise their resource properties, both static and dynamic, such as available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a resource offer ad. A user specifies a resource request ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. HTCondor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, HTCondor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

3.2.3 Installation and Configuration

The HTCondor configuration files are used to customize how HTCondor operates at a given site. The basic configura- tion as shipped with HTCondor works well for most sites. Each HTCondor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used, including pool-wide, platform-specific, and machine-specific configuration files. Environment variables may also contribute to the configuration. The result of configuration is a list of key/value pairs. Each key is a configuration variable name, and each value is a string literal that may utilize macro substitution (as defined below). Some configuration variables are evaluated by HTCondor as ClassAd expressions; some are not. Consult the documentation for each specific case. Unless otherwise noted, configuration values that are expected to be numeric or boolean constants may be any valid ClassAd expression of operators on constants. Example:

```
MINUTE = 60
HOUR = (60 * \$(MINUTE))
SHUTDOWN\_GRACEFUL\_TIMEOUT = (\$(HOUR)*24)
```

I was tasked with installation and configuration of HTCondor job submission platform. Installation of HTCondor is pretty straightforward using the **aptitude** package manager. Following commands achieve this task.

```
    sudo apt-get update
    sudo apt-get update
    sudo apt-get install condor
```

Manager Configuration The configuration for the manager is specified in /etc/condor/config.d/condor_config.local file.

```
DAEMON_LIST        = COLLECTOR, MASTER, NEGOTIATOR
ALLOW_CONFIG       = root@alice.icts.res.in/\$(IP_ADDRESS),root@crm.
    alice.icts.res.in
ALLOW_WRITE        = *

SEC_PASSWORD_FILE = /etc/condor/secret/pool_password

SEC_DAEMON_AUTHENTICATION              = REQUIRED
SEC_DAEMON_INTEGRITY                   = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = PASSWORD
ALLOW_DAEMON                           = condor_pool@*

SEC_NEGOTIATOR_AUTHENTICATION                = REQUIRED
SEC_NEGOTIATOR_INTEGRITY                     = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS        = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS            = FS, PASSWORD, KERBEROS,
    GSI
ALLOW_NEGOTIATOR                             = condor_pool@*

SEC_ADVERTISE_STARTD_AUTHENTICATION                  = REQUIRED
SEC_ADVERTISE_STARTD_INTEGRITY                       = REQUIRED
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS          = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS                    = FS, PASSWORD,
    KERBEROS, GSI
ALLOW_ADVERTISE_STARTD                               = condor_pool@*

SEC_ADVERTISE_MASTER_AUTHENTICATION                  = REQUIRED
SEC_ADVERTISE_MASTER_INTEGRITY                       = REQUIRED
SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS          = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS                    = FS, PASSWORD,
    KERBEROS, GSI
ALLOW_ADVERTISE_MASTER                               = condor_pool@*

SEC_ADVERTISE_SCHEDD_AUTHENTICATION                  = REQUIRED
SEC_ADVERTISE_SCHEDD_INTEGRITY                       = REQUIRED
SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS          = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS                    = FS, PASSWORD,
    KERBEROS, GSI
ALLOW_ADVERTISE_SCHEDD                               = condor_pool@*

SEC_ADMINISTRATOR_AUTHENTICATION                     = REQUIRED
SEC_ADMINISTRATOR_INTEGRITY                          = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS             = PASSWORD, FS
ALLOW_ADMINISTRATOR                                  = condor_pool@alice
    .icts.res.in/\$(IP_ADDRESS),\ condor_pool@crm.icts.res.in/\$(
    IP_ADDRESS)
```

Submit node Configuration The configuration for the submit machines is specified in /etc/condor/config.d/condor_config.local file.

```
# HTCondor configuration file

CENTRAL_MANAGER1 = 10.0.0.100
CONDOR_HOST = \$(CENTRAL_MANAGER1)
DAEMON_LIST = MASTER, SCHEDD
ALLOW_CONFIG = root@mydomain/\$(IP_ADDRESS)
ALLOW_WRITE = *

SEC_PASSWORD_FILE = /etc/condor/secret/pool_password

SEC_DAEMON_AUTHENTICATION = REQUIRED
SEC_DAEMON_INTEGRITY = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_DAEMON = condor_pool@*

SEC_NEGOTIATOR_AUTHENTICATION = REQUIRED
SEC_NEGOTIATOR_INTEGRITY = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = PASSWORD
ALLOW_NEGOTIATOR = condor_pool@*

SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
```

Compute node Configuration The configuration for the submit machines is specified in /etc/condor/config.d/condor_config.local file.

```
CENTRAL_MANAGER = crm.alice.icts.res.in
CONDOR_HOST     = \$(CENTRAL_MANAGER)

DAEMON_LIST       = MASTER, STARTD
SEC_PASSWORD_FILE = /etc/condor/secret/pool_password
ALLOW_WRITE       = *

SEC_DAEMON_AUTHENTICATION         = REQUIRED
SEC_DAEMON_INTEGRITY              = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_DAEMON                      = condor_pool@\$(UID_DOMAIN)/* \
                                        condor@\$(UID_DOMAIN)/\$(
    IP_ADDRESS)

SEC_NEGOTIATOR_AUTHENTICATION         = REQUIRED
SEC_NEGOTIATOR_INTEGRITY              = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS,
    GSI
ALLOW_NEGOTIATOR                      = condor_pool@\$(UID_DOMAIN
    )/* \
                                                          condor@\$(
    UID_DOMAIN)/\$(IP_ADDRESS)

SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_CONFIG = root@\$(UID_DOMAIN), root@crm.alice.icts.res.in
```

3.2.4 Security

Security in HTCondor is a broad issue, with many aspects to consider. Because HTCondor's main purpose is to allow users to run arbitrary code on large numbers of computers, it is important to try to limit who can access an HTCondor pool and what privileges they have when using the pool.

There is a distinction between the kinds of resource attacks HTCondor can defeat, and the kinds of attacks HTCon- dor cannot defeat. HTCondor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. HTCondor does not run user jobs in sandboxes (standard universe jobs are a partial exception to this), so HTCondor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. HTCondor assumes that users are trustworthy. HTCondor can prevent unauthorized access to the HTCondor pool, to help ensure that only trusted users have access to the pool. In addition, HTCondor provides encryption and integrity checking, to ensure that data (both HTCondor's data and user jobs' data) has not been examined or tampered with while in transit.

At the heart of HTCondor's security model is the notion that communications are subject to various security checks. A request from one HTCondor daemon to another may require authentication to prevent subversion of the system. A request from a user of HTCondor may need to be denied due to the confidential nature of the request. The security model handles these example situations and many more.

The password method provides mutual authentication through the use of a shared secret. The shared secret in this context is referred to as the pool password. Before a daemon can use password authentication, the pool password must be stored on the daemon's local machine. On Unix, the password will be placed in a file defined by the configuration variable SEC_PASSWORD_FILE. This file will be accessible only by the UID that HTCondor is started as(root). Under Unix, the password file can be generated by using the following command to write directly to the password file:

```
condor_store_cred -f /path/to/password/file
```

In addition, storing the pool password to a given machine requires CONFIG-level access. For example, if the pool password should only be set locally, and only by root, the following would be placed in the global configuration file.

```
ALLOW_CONFIG = root@mydomain/\$(IP_ADDRESS)
```

3.3 User priorities and job policy

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time, meaning during negotiation and matchmaking. Therefore, there are ClassAd attributes that take on defined values only during negotiation, making them ephemeral. In addition to allocation, HTCondor may preempt a machine claim and reallocate it when conditions change.

Too many preemptions lead to thrashing, a condition in which negotiation for a machine identifies a new job with a better priority most every cycle. Each job is, in turn, preempted, and no job finishes. To avoid this situation, the PREEMPTION_REQUIREMENTS configuration variable is defined for and used only by the condor_negotiator daemon to specify the conditions that must be met for a preemption to occur. When preemption is enabled, it is usually defined to deny preemption if a current running job has been running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval. Note that PREEMPTION_REQUIREMENTS only applies to preemptions due to user priority. It does not have any effect if the machine's RANK expression prefers a different job, or if the machine's policy causes the job to vacate due to other activity on the machine.

Priority calculation HTCondor's priority calculation algorithm is given below.

The Real User Priority(RUP) of a user is given by.

$$\pi_r(u,t) = \beta * \pi(u, t - \Delta t) + (1 - \beta) * \rho(u,t) \tag{1}$$

The Effective User priority is given by

$$\pi_e(u,t) = \pi_r(u,t) * f(u,t) \tag{2}$$

$f(u,t)$ is the priority boost factor for user.

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of β ensures that the calculation of $r(u,t)$ can be calculated over non-uniform time intervals Δt without affecting the calculation. The time interval Δt varies due to events internal to the system, but HTCondor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

4 IO Benchmarking

I performed IO Benchmarking for various RAID configurations of the storage server. These test were performed using python benchmarking suite.

IO Performance for ext4 filesystem

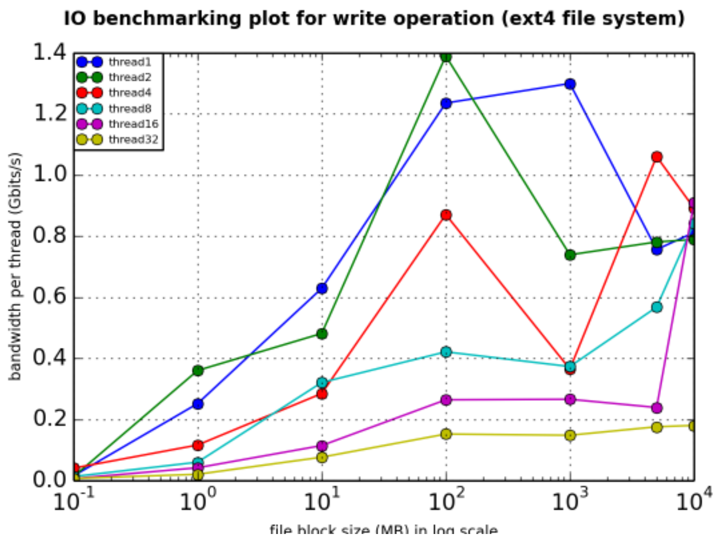

Figure 2: Write performance for ext4

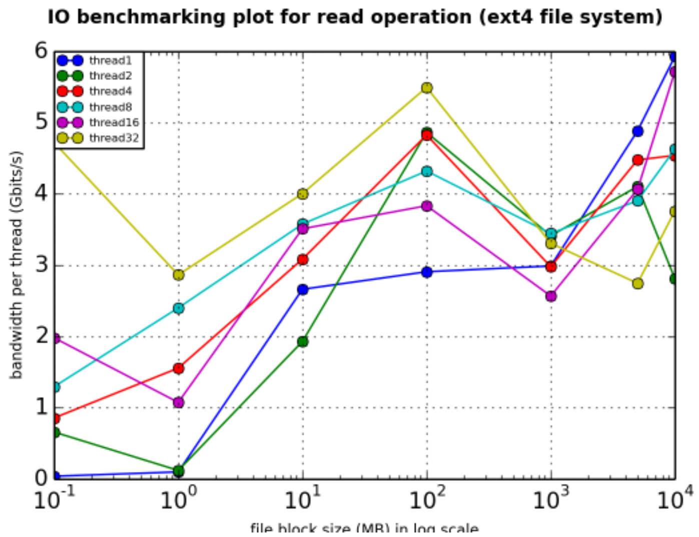

Figure 3: Read performance for ext4

IO Performance for ZFS RAIDz2 pool

Result: ZFS RAIDz2 pool with SSD cache gave the best performance and hence it was used for the storage server.

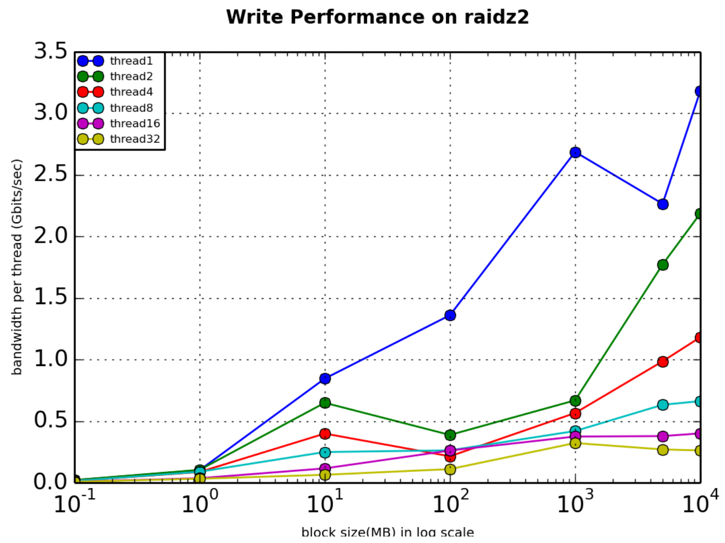

Figure 4: Write performance for ZFS RAIDz2

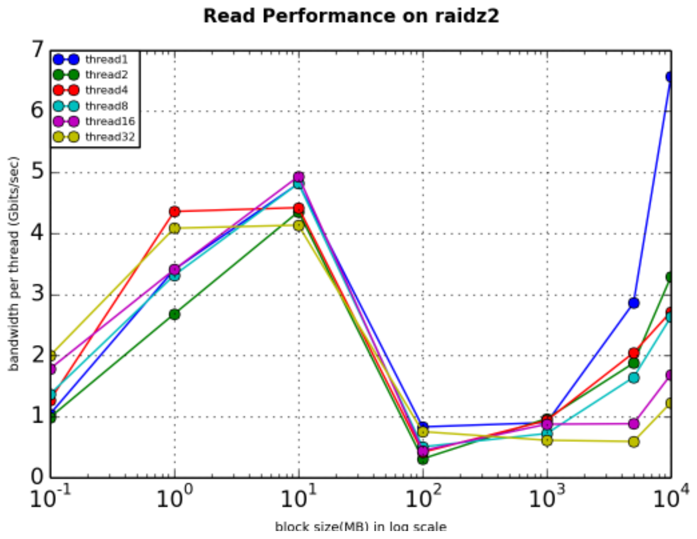

Figure 5: Read performance for ZFS RAIDz2

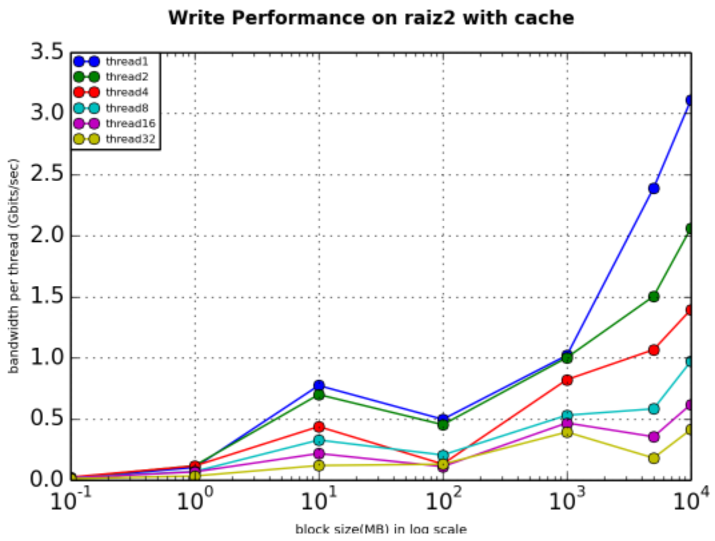

Figure 6: Write performance for ZFS RAIDz2 with cache

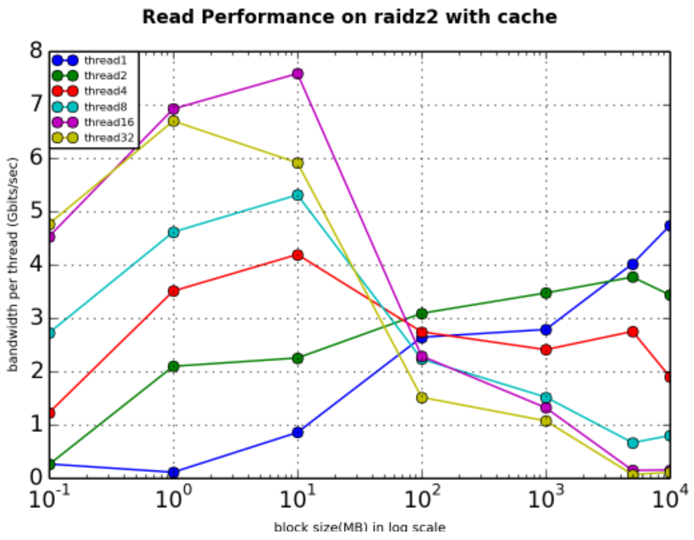

Figure 7: Read performance for ZFS RAIDz2 with cache