CS 6210: Project 2
Simulating the Xen Split Driver Model

Shubhojit Chattopadhyay(ssc3@gatech.edu) [GTID: 902694799]
Abhinav Narain(nabhinav3@mail.gatech.edu) [GTID: 98401495]

I. Description of Xen Hypervisor

Hypervisor or Virtual machine monitors are used to utilize the system resources to the maximum by allowing multiple operating systems run over a thin layer of hardware abstraction.

They are two types of Virtualization:
1) Para Virtualization
2) Full Virtualization

In para virtualization, parts of the host operating system are modified so that the calls to those low level functionalities can be trapped by the hypervisor.

In Full Virtualization, the host operating systems run unmodified on the hypervisor.

Xen Hypevisor lies in the category of Para Virtualized hypervisors. Here we elaborate the details of how the Hypervisor interacts/communicates with the host operating systems. The two mechanisms for control interactions btween Xen and an overlying domain are synchronous calls from a domain to Xen (hypercalls) and notifications to domain from Xen are done using an asynchronous event mechanism.

The first case allows the VM to do a synchronous software trap into the Hypervisor to perform a priviledge operation.

The second case is when events are used to notify a task is over. Pending events are kept in a per-domain bitmask handler which is updated by Xen before invoking an event-callback handler specified by the guest OS.

About the Data transfer, primarily I/O rings are used. The important motivation being resource management and event notification. One needs to minimize the time in demultiplexing of data on interrupt request. The overhead of managing the buffers is carried is deferred to the
stage when computation may be accounted to a particular domain. I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

The I/O is done using a ring containing the I/O descriptors which are accessible from within Xen. These descriptors access the data region indirectly, allocated in the guest OS. The I/O requests can be served in random order according to the scheduling properties of the Xen Scheduler.

In this project we have simulated this particular part of Xen in user space.

II. Related source of Xen

a. Checking if there is any place in the ring:

```
int mem_event_check_ring(struct domain *d)
{
    struct vcpu *curr = current;
    int free_requests;
    int ring_full;

    if ( !d->mem_event.ring_page )
        return -1;

    mem_event_ring_lock(d);

    free_requests = RING_FREE_REQUESTS(&d->mem_event.front_ring);
    if ( unlikely(free_requests < 2) )
    {
        gdprintk(XENLOG_INFO, "free request slots: %d\n", free_requests);
        WARN_ON(free_requests == 0);
    }
    ring_full = free_requests < MEM_EVENT_RING_THRESHOLD ? 1 : 0;

    if ( (curr->domain->domain_id == d->domain_id) && ring_full )
    {
        set_bit(_VPF_mem_event, &curr->pause_flags);
        vcpu_sleep_nosync(curr);
    }

    mem_event_ring_unlock(d);
```

```
    return ring_full;
}
```

b. Putting into ring buffer

```
void mem_event_put_request(struct domain *d, mem_event_request_t *req)
{
    mem_event_front_ring_t *front_ring;
    RING_IDX req_prod;

    mem_event_ring_lock(d);

    front_ring = &d->mem_event.front_ring;
    req_prod = front_ring->req_prod_pvt;

    /* Copy request */
    memcpy(RING_GET_REQUEST(front_ring, req_prod), req, sizeof(*req));
    req_prod++;

    /* Update ring */
    front_ring->req_prod_pvt = req_prod;
    RING_PUSH_REQUESTS(front_ring);

    mem_event_ring_unlock(d);

    notify_via_xen_event_channel(d, d->mem_event.xen_port);
}
```

c. Getting response from ring buffer

```
void mem_event_get_response(struct domain *d, mem_event_response_t *rsp)
{
    mem_event_front_ring_t *front_ring;
    RING_IDX rsp_cons;

    mem_event_ring_lock(d);

    front_ring = &d->mem_event.front_ring;
    rsp_cons = front_ring->rsp_cons;

    /* Copy response */
    memcpy(rsp, RING_GET_RESPONSE(front_ring, rsp_cons), sizeof(*rsp));
    rsp_cons++;

    /* Update ring */
```

```
    front_ring->rsp_cons = rsp_cons;
    front_ring->sring->rsp_event = rsp_cons + 1;

    mem_event_ring_unlock(d);
}
```

III. Architecture Design

a. Client-Server Model using shared memory

The typical structure of code used to create client-server model using shared memory is as follows:

```
/* Locate the Physical memory segment */
shmid = shmget(key, SHMSZ, 0666);

/* Attach own address space to this physical memory */
shm = shmat(shmid, NULL, 0);

/* Write into/Read from attached memory segment */
memcpy(shm, buf, 512);

/* Detach memory segment */
shmdt(shm);

/*Delete physical memory segment */
shmctl(shmid, IPC_RMID, NULL);
```

The shared memory was created with simple allocation of 2 free physical memory locations and mapping them to the client and server address spaces. One of the two shared memory location is used for each client to notify the server of it's arrival and the other shared space is the actual ring buffer. The following diagram illustrates the shared memory architecture:

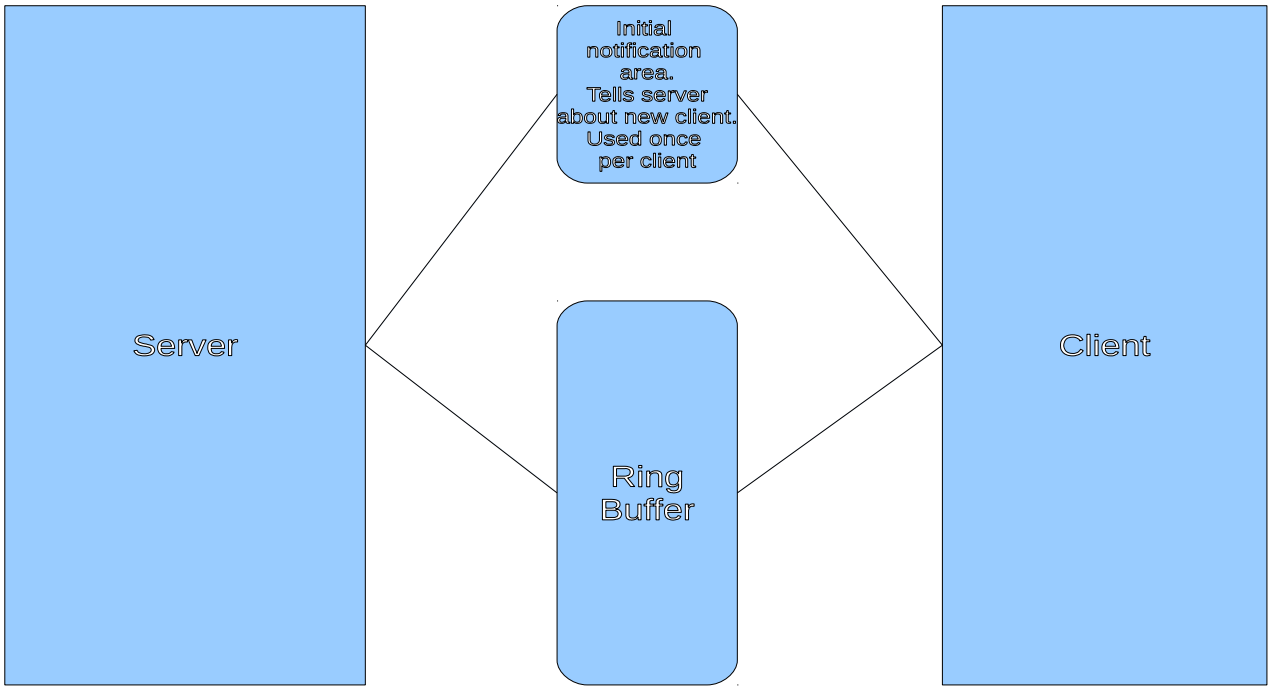


Figure 1: Communication over multiple shared areas

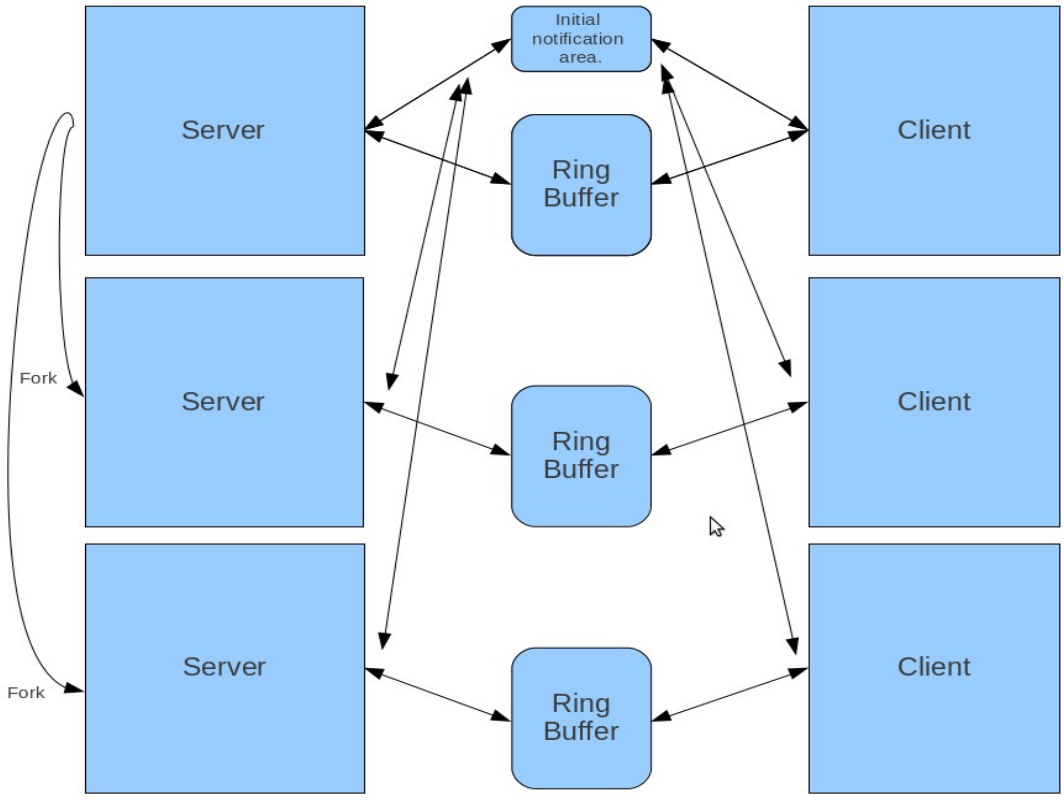


Figure 2: How multiple shared areas are used by multiple client-servers

As we can see, the one notification area is shared by all clients. All the clients have this notification area's shared memory key given to them statically. When a new client arrives, it sends a random key to the server through this shared area. Once the server acknowledges the presence of this client, it forks and creates a new process to handle this client, with a new shared memory for the ring buffer using the key given by the client.

b. Design of Ring buffer

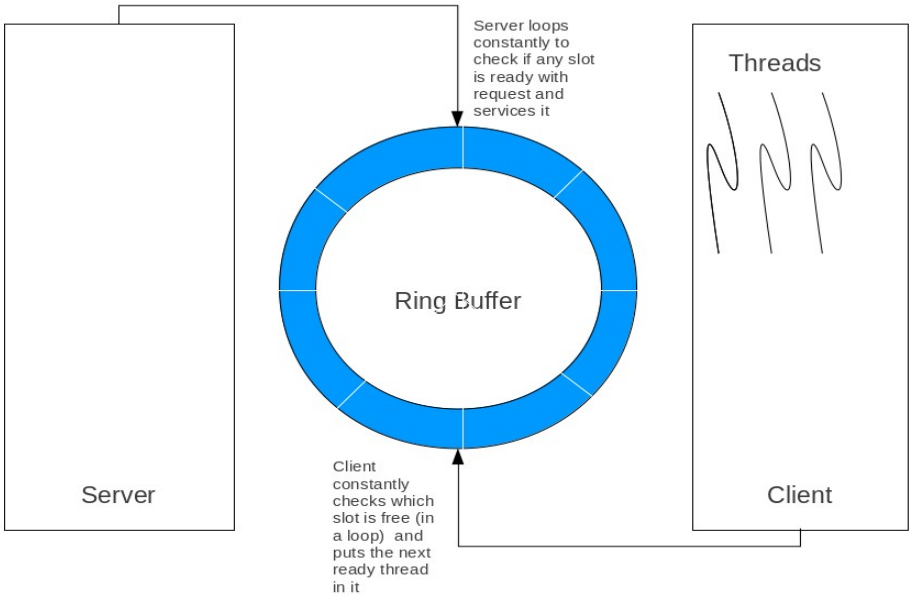
The ring buffer design is a struct of structs as shown below:

```
#define NUM_SLOTS 100

typedef struct _slot{
  int preamble;
  int tid;
  int global_req_id;
  int sector_number;
  int slot_id;
  int data_ready;
  int full;
  struct timespec start_time, end_time;
  unsigned char data[512];
} slot_t;

typedef struct _device_ring{
  int num_of_sectors;
  slot_t slot[NUM_SLOTS];
  int num_of_sectors_ready;
} device_ring;
```

The implementation is slightly different from the diagram showed in the problem statement. It can be visualized as follows:



Note that there is no rotation of the buffer itself. Rather, the server iterates through each slot to see if any slot is ready with a request and services it, while the client iterates through each slot to see if it's free and tries to put a new request in it

c. Design of Daemon Servers:

The main server is supposed to process each incoming client equally and ensure that there is no interference in data between multiple clients. In order to do this, the server is forked each time a new client comes in.

```
While(1)
{
   key = get_client(); //Loops to check for a new client and returns the key of the new client

  if (key!=0)
  {
     switch(fork)
     {
        case 0:
            //Do the ring buffer allocation and sector servicing
```

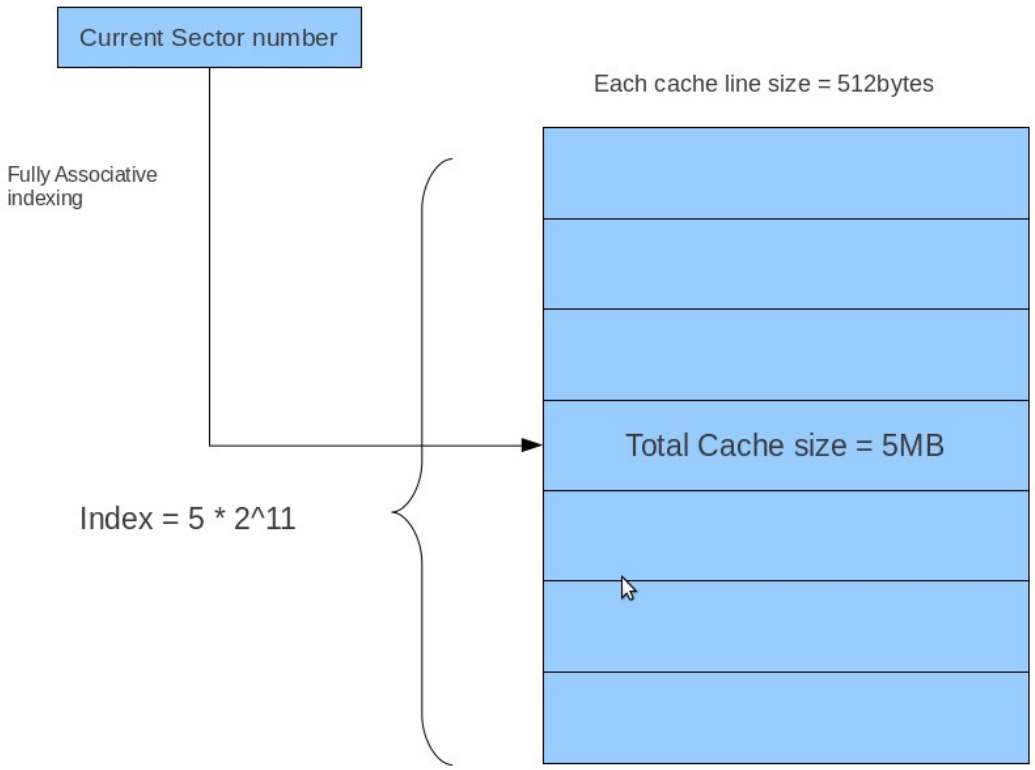
d. Cache Architecture

A small cache was designed to hold frequently used disk sectors continuously. Whenever a request is made, we know the sector number that the client is looking for. This sector number is used to index into the cache and pick the next 512 bytes. The cache is dynamically allocated initially and memory aligned (posix_memalign) so that each cache line holds exactly 512 bytes. The cache is fully-associative.

The size of the whole cache = 0.01* 512Mb (the maximum value of dd) = 5 Mb
So, Number of indexes into cache (num_of_sectors) = Total Size of cache / Size of each cache line
= (5 * 2^20) / 512
= 5 * 2^11

The cache works if we do not use srand() with rand(). Rand() tends to return the same sector numbers often. This is particularly helpful with a huge disk image.

The cache diagram is shown below:



e. Reading data from the disk

The data is read from the disk in O_DIRECT form. This format gives data directly byte by byte instead of buffering data. This immensely slows down the reading process. Also, to read in O_DIRECT format, the data needs to be aligned in memory, so that all the 512 bytes are fetched and copied into a buffer in our code.

The code for reading data is as follows:

```
unsigned char* buf;
posix_memalign((void*) &buf, 512, 4096);

#ifdef CACHE_ENABLED
unsigned char* buffer_cache;

posix_memalign((void*) &buffer_cache, 512, 512);
buffer_cache = lookup_in_cache(sector_num);
int j;
if(buffer_cache != NULL)
{
        memcpy(_shm->slot[i].data, buffer_cache, 512);
}
else
{
#endif
        int fd = open("disk1.img", O_DIRECT | O_SYNC);
        int offset = lseek(fd, sector_num*512, SEEK_SET);
        read(fd, /*_shm->slot[i].data*/ buf, 512);
        memcpy(_shm->slot[i].data, buf, 512);
        insert_in_cache(buf, sector_num);

        close(fd);
#ifdef CACHE_ENABLED
}
#endif
```

f. Maintaining fairness amongst multiple Threads and Clients

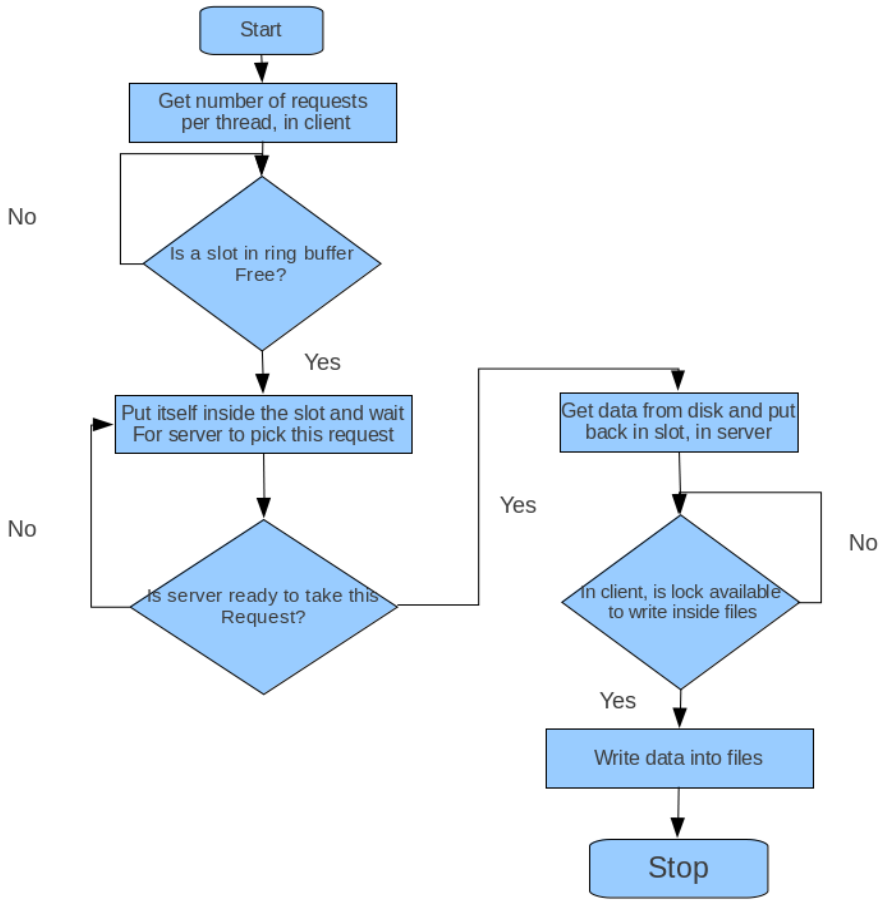
Multiple threads are all allowed to be spawned serially using pthread library's traditional thread spawning mechanism. Once the threads have been spawned, they update their own values and spin over the ring buffer mutex lock to see if any slot is available. If there is a slot available, the thread takes the mutex and updates the ring buffer and then releases it until data arrives back from the server. Once this is released, there is contention amongst the remaining threads to acquire the lock and insert their requests into the ring buffer.

Multiple clients are all run in quick succession. Each client is a separate process and the server is forked to handle each of these clients separately. So there is no contention between clients in terms of server address space. The servicing of new clients is as fair as kernel scheduler fairness of the operating

system on which this program is running.

g**. Lifecycle of one request**

The following diagram explains what happens with a request from start to finish.



III. Performance Optimization

a. Use of multiple locks

The three places where there could have been any multithreading conflict are:

- When updating the number of requests each thread has to handle
- When a thread is allocating a request into one slot
- When the data is received back and needs to be written into files

All these three things can be combined into a single critical section lock. But that would degrade

performance a lot. A more deeper granularity lock system gives better performance.

Note that there is no lock required for server since as soon as server gets a slot, it takes the sector number from that slot and inserts data back into the slot. Until then, that slot is blocked and no one else can use it.

b. Caching

As explained above, the cache mechanism helps in fast indexing. However, the size of the cache is too small. Also, caches work because of principle of locality (spatial and temporal). However, in this case, the sector numbers in each request are generated randomly. So there is no locality whatsoever. Hence the caches perform poorly.

c. Forking of server

Each time a new client arrives, the parent server detects and forks itself. This means that for each client-server pair, there is a separate address space for the server and a separate shared ring buffer. So, this results in faster performance and data protection.

d. Potential Optimization (not implemented)

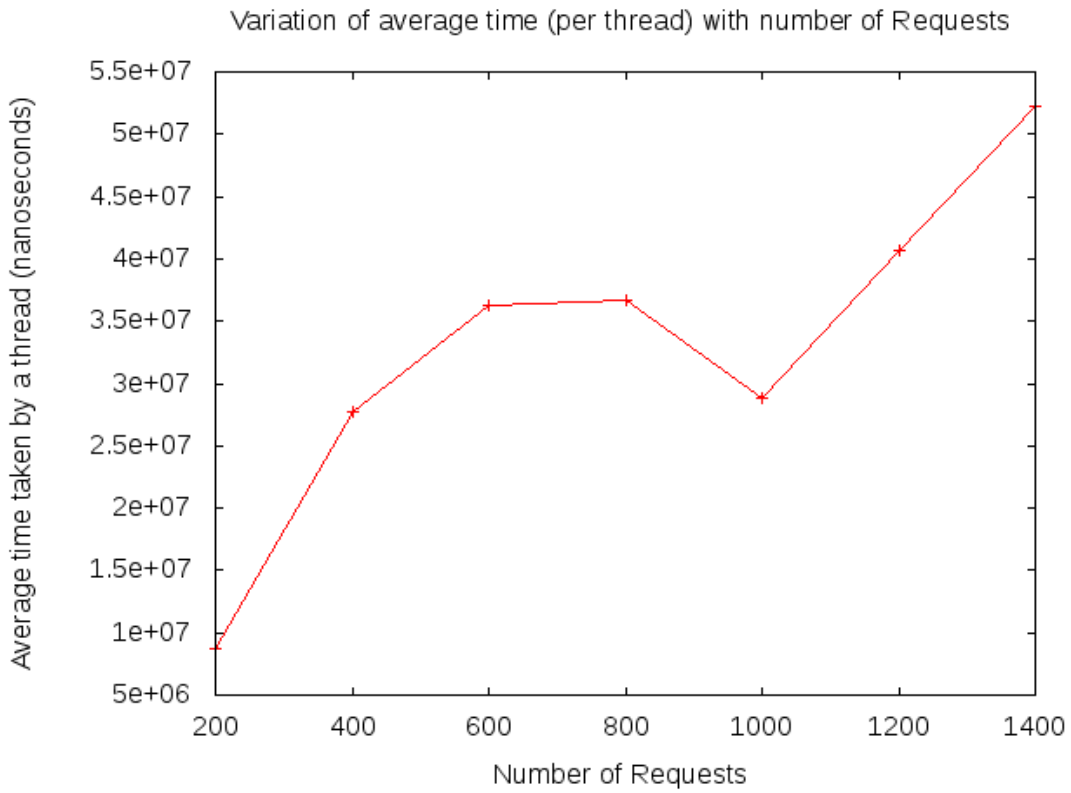
One potential optimization to this approach is the reduction of multiple spinlocks. This is possible if each thread simply dispatches the sector number request into the ring buffer and dies. That means, the threads will not block/spin. The data that eventually arrives will be serviced by the Client itself rather than each thread of the client.

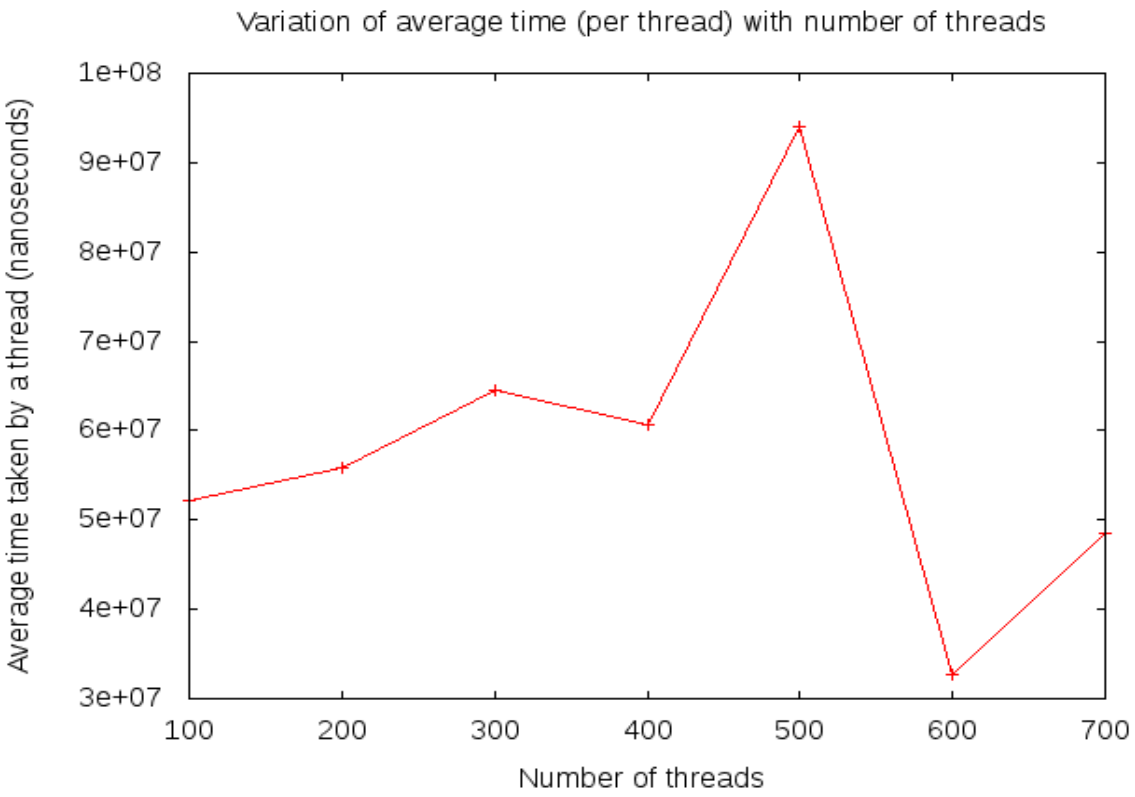
However, this was not implemented since in the problem statement, it was asked that each thread should block until the data is fully available.

IV. Results

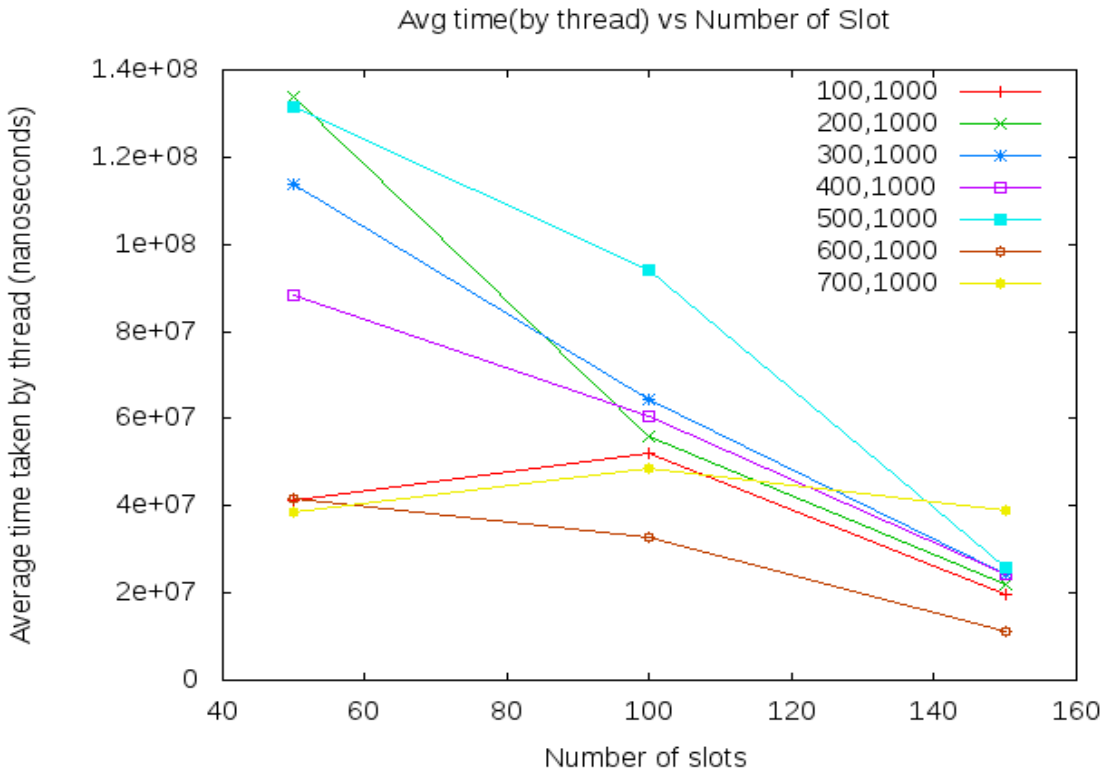
For each of these observations, the following values are used unless mentioned otherwise:
The number of slots = 100
Number of clients = 5
Number of threads = 10
Number of requests = 1000
Disk Size : X=1M, Y=500

The following section presents the performance results of the Xen driver model we simulated.
There is a general trend for the service time per thread increases as the number of requests are increased . This is because of the fact that the number of seeks needed to be done for more requests are more.

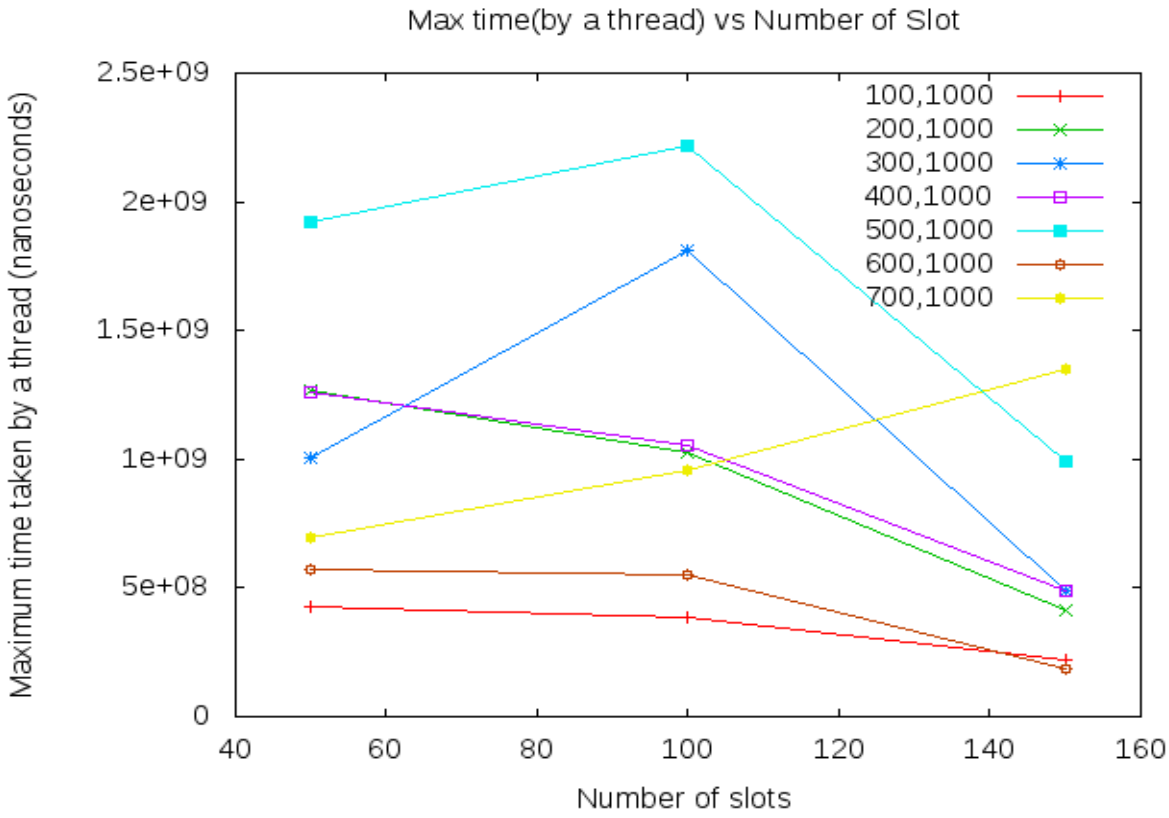




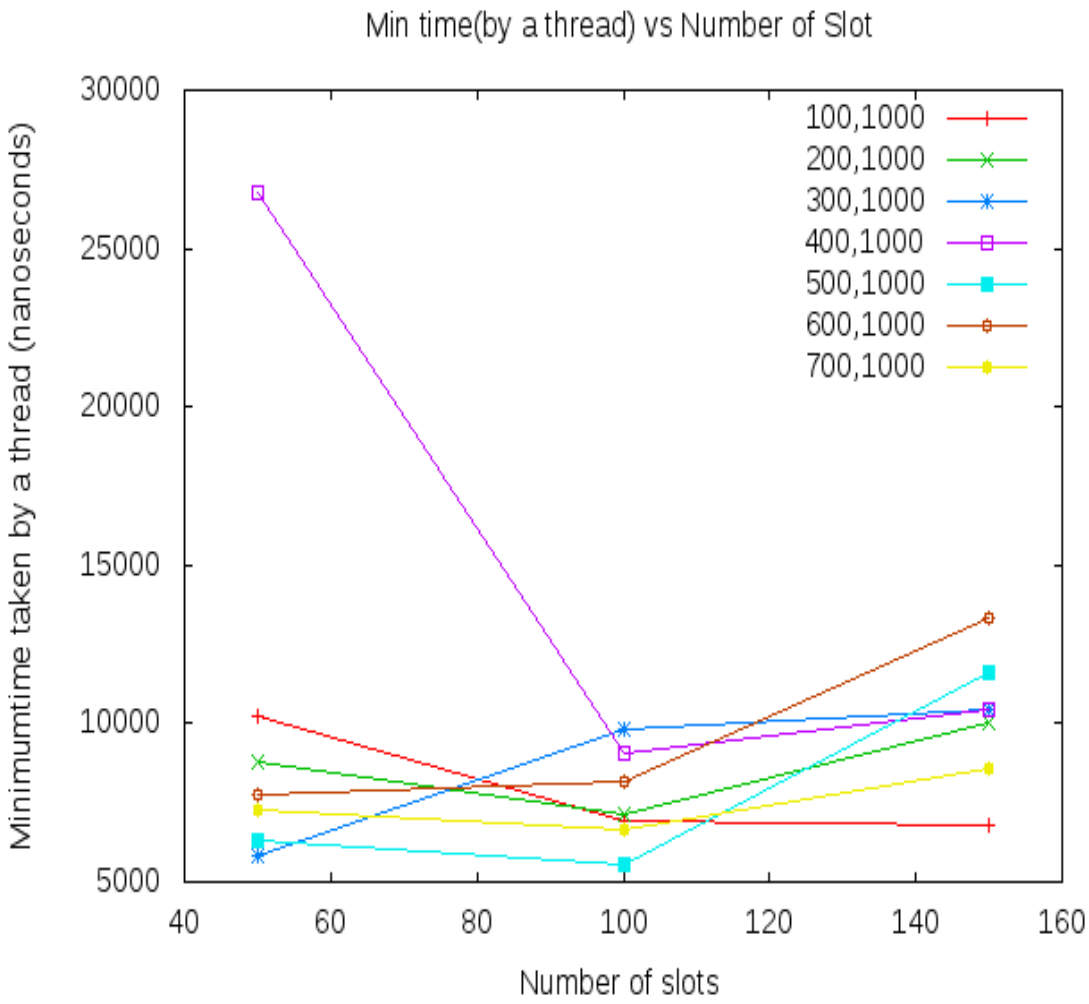
The average time taken by a thread shows no particular behavior in our case. The trend should be decreasing intuitively, but it shows a sharp peak at 500. The overheads due to context switching might not be negligible and switching might be more frequent than required.



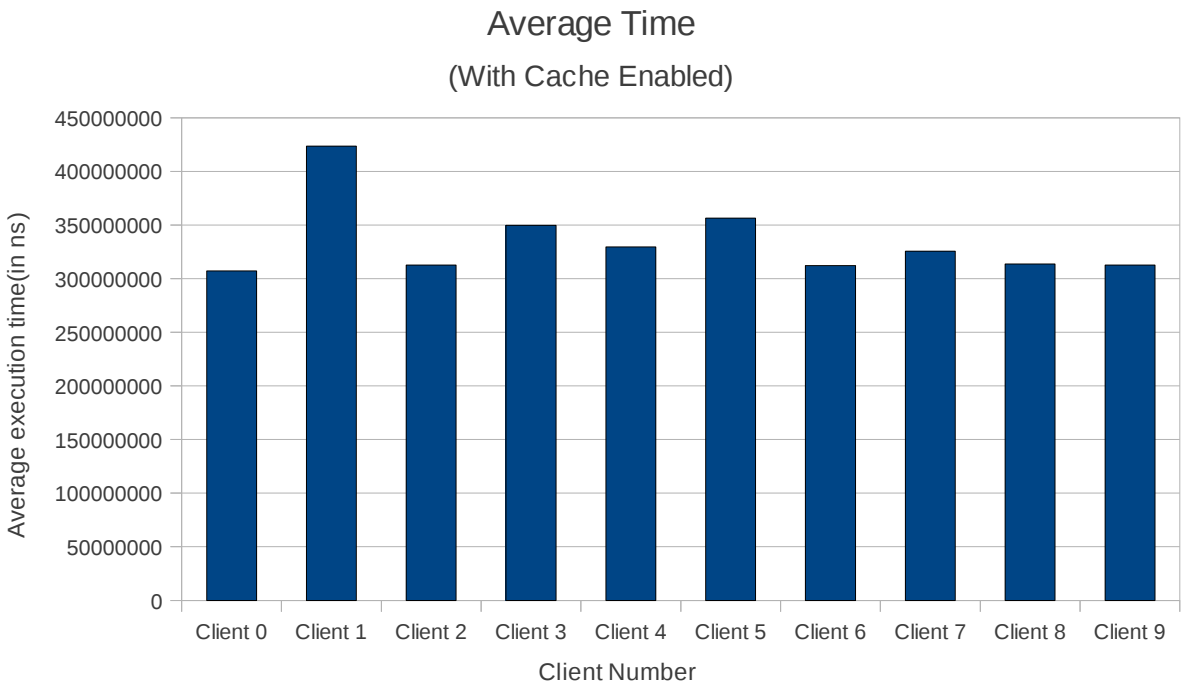
The above graph shows the average time taken by a thread when the number of slots increases from 50 to 150. Each line represents the a tuple of number of threads and num of requests. We can notice, a general trend that the average time decreases with the increase in number of plots, apart from two extreme cases when the number of threads is 100 and 700.



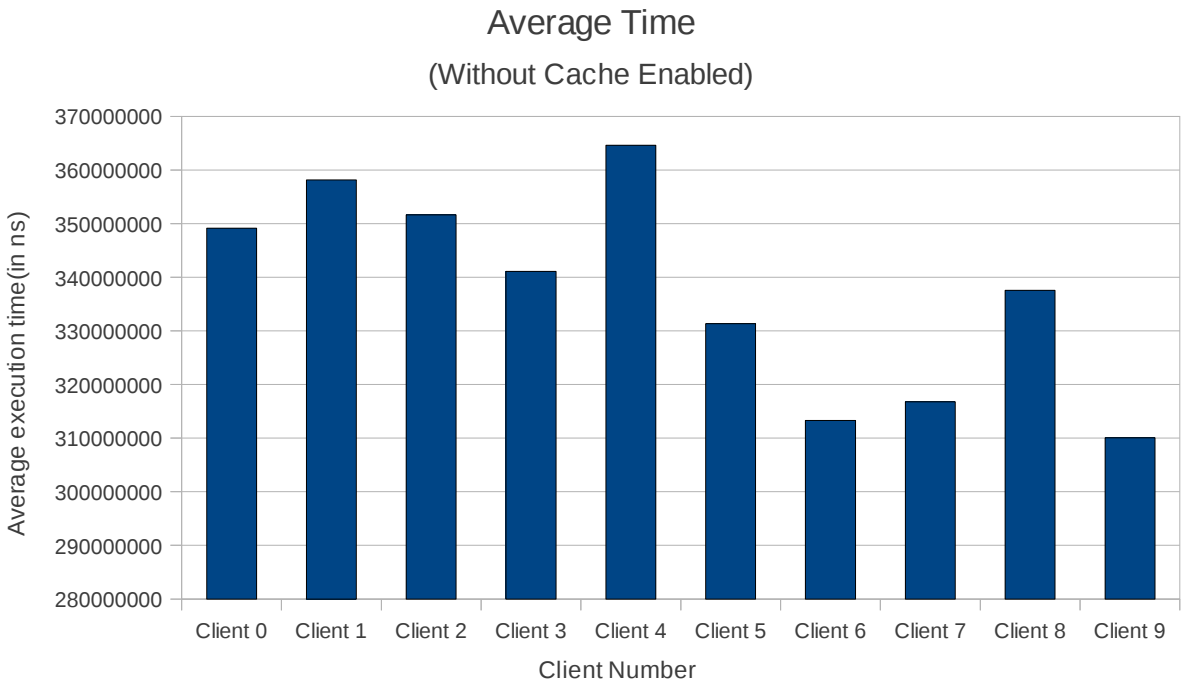
For a particular number of thread count, we see the maximum time spent by the thread decrease with the increas in the number of slots. It can also be inferred that the number of threads are varied (that is looking across different lines on the graph), there is no usual correlation with the numer of threads and the number of slots.



When the number of slots is 100, we can notice all the threads take least time to work, while . Its can also be seem, time taken by few threads also hits a maximum at this value of slots count, hence we cannot say with confidence that this is an optimal number.
The number would be dependent on the architecture in general.

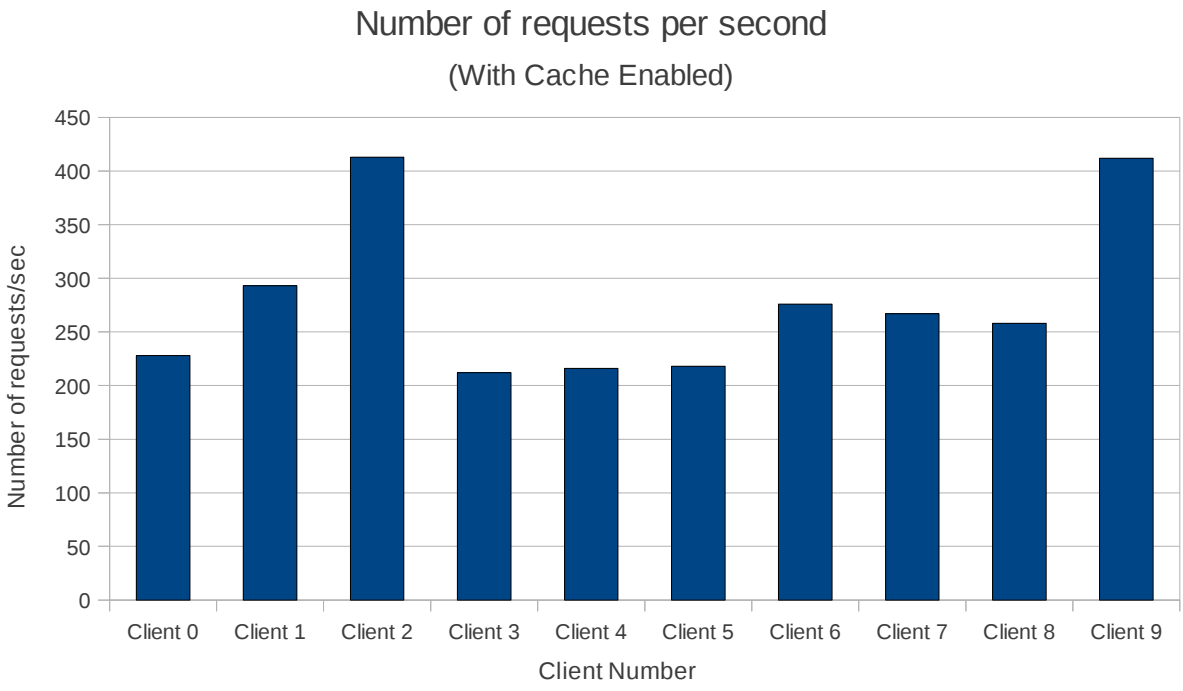


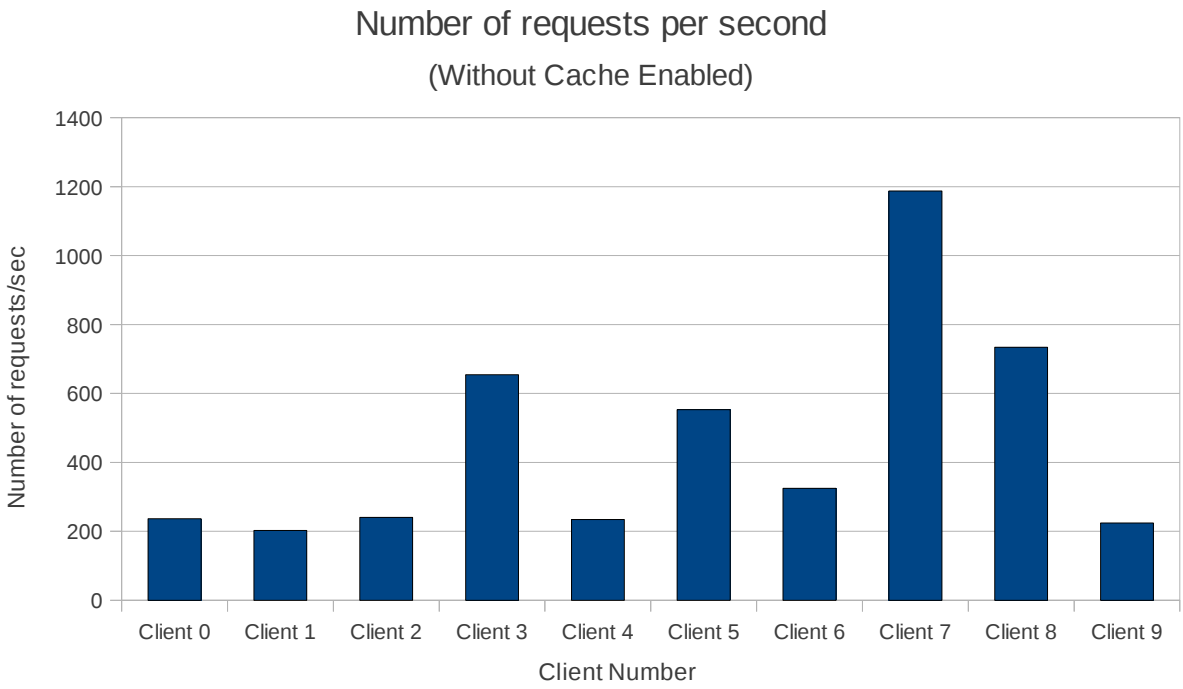
The average time for 10 clients with cache enabled is: 334377236



The average time for 10 clients with cache enabled is: 337367226

Note: The average execution time is quite stable and fair across multiple clients


Number of requests per second
(With Cache Enabled)
Number of requests/sec
450
400
350
300
250
200
150
100
50
0
Client 0
Client 1
Client 2
Client 3
Client 4
Client 5
Client 6
Client 7
Client 8
Client 9
Client Number


Number of requests per second
(Without Cache Enabled)
Number of requests/sec
1400
1200
1000
800
600
400
200
0
Client 0
Client 1
Client 2
Client 3
Client 4
Client 5
Client 6
Client 7
Client 8
Client 9
Client Number

V. Regression and testing methodology

The implementation required constant testing to check the integrity of the results and fast testing methodology. Two scripts were written to do this

- Client-regression.sh: This was a list of multiple client requests with varying number of requests and number of threads
- checker.pl: This script ran another C code to check the sector number in Sectors.PID and then extract the 512 bSearchytes related to that sector from read.PID. Then it would compare all these 512 bytes with the corresponding sector number in disk1.img

VI. Failure Cases

A potential problem with current blocking thread approach is that if there are too many threads created, the contention for critical section is very high. If there are 10 clients, each with 1000 threads, that meant there are potentially 10000 threads spinning at a particular instant. This can be solved by the **Potential Optimization** technique mentioned above. However, the current problem statement given limits this optimization.

VII. Role of Team Members

Shubhojit Chattopadhyay:

- Initial exploration
- Designed and wrote the Ring buffer structure
- Designed and wrote the Sector number and actual data sharing mechanism in shared mem
- Investigated O_DIRECT and wrote code for File reading in Server and writing in client
- Designed and wrote the cache
- Wrote scripts for checking the correctness and regressions
- Wrote part of the report

Abhinav Narain:

- Initial exploration
- Wrote code for shared memory creation
- Wrote code for initial Server-Client design and initial creation of daemons
- Investigated potential issues with shared memory and solutions to them
- Debugged slightly buggy operations during the course of the project
- Collected statistics using tests and checked them for correctness
- Wrote part of the report

VII. References

[a] **Xen and Art of Virtualization** - Paul Barham, Boris Dragovic, Kier Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield

[b] http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Fapis%2Fapiexusmem.htm

[c] Xen source: http://xen.org/products/downloads.html

[d] Man pages