Optimal Grouping of Cores in BOSS MOOL

A Project Report

submitted by

ABHIJITH C S

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2015

THESIS CERTIFICATE

This is to certify that the thesis titled **Optimal Grouping of Cores in BOSS MOOL**, submitted by **Abhijith C S**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. D Janakiram
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT Madras, 600 036

Place : Chennai

Date :

ACKNOWLEDGEMENTS

I would like to thank my project guide and faculty advisor, Dr. D Janakiram, for all his supervision and help during the project work. I would also like to thank the people at Distributed & Object Systems Lab - Aditya Sapate (Dual Degree), Abhinay Bulakh (MS Scholar), Mahender (M-Tech) and Jaya Durga (Developer from CDAC) for helping me out at different stages of my project work. Lastly, I thank my friends Amal Joy, Nandakishore M M, Mohammed Shafeeq E T, Anas Jafry, Hamdan M Ridwan and Aslamah Rahman and everyone else who supported me in one way or the other during my four years of life at IIT Madras.

ABSTRACT

KEYWORDS: Core Grouping, BOSS MOOL, Linux Kernel

In multi-core systems, the scheduling policy has a very important role to play in achieving maximized performance. But at the same time, the improper and unbalanced assignment of processes to the cores might degrade the performance. By default, all the cores in a multi-core system are assumed to be of equal performance and computing capacity. Any core can get assigned with any tasks by the scheduler irrespective of its nature. Only priority of the tasks are counted. It is an attempt to classify the cores into different sets based on certain criteria, so that each set of cores will get assigned only with a specific type of tasks. Dedicating each set of cores to perform specific types of tasks might improve the overall performance.

This research work attempts to suggest an optimal way of grouping cores based on certain criteria. There are three possible ways for grouping the cores from two different perspectives, proposed here - from cores' perspective (cores blocking specific type of processes from accessing it) and from processes' perspective (restricting the processes from getting scheduled on a predefined set of cores). The later method has lesser number of computations and which is concluded as optimal and generic solution. Irrespective of the criteria for classification, the way in which the cores are grouped affects the performance. The optimal ratio to group the cores could be based on the workload and which could be calculated dynamically.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BOSS	Bharat Operating System Solution
MOOL	Minimalistic Object Oriented Linux
CPU	Central Processing Unit
GPU	Graphics Processing Unit
OS	Operating System
RB Tree	Red-Black Tree
HW-Thread	Hardware Thread
CFS	Completely Fair Scheduler

CHAPTER 1

INTRODUCTION

In order to achieve the best performance out of the available computing resources, it is very important that the resources are to be well utilised. Core is the basic computation unit of the CPU. It can run a single program context or multiple ones. A CPU may have one or more cores to perform tasks at a given time.

There could be many threads to be run, but at a given time the CPU can only run a limited number of threads which is equal to, *Number of Cores* $\times$ *Number of HW-threads per Core*. The rest of the threads would have to wait for the OS to schedule them whether by preempting currently running tasks or any other means. An efficient scheduler equally and evenly distributes the threads over the cores by considering several factors like how prior the tasks are, how long it has been waiting etc. There are quite a few effective schedulers designed for Linux. By default the scheduling algorithms assumes all the cores to be of equal computing power and performance. It may not be the case always.

It is an attempt to study how categorizing of the cores help to improve the performance. The available cores could be grouped in different ways with respect to its properties. Say for example, if a total of 8 cores are available, 4 could be dedicated for processing the real-time tasks and 4 for the other tasks. The idea is to find a novel criterion for such classification of groups in BOSS MOOL, which might improve the performance.

In multi-core systems, though the scheduling policy has got the very important role to play, the inefficient or unbalanced assignment of processes to the cores might cause poor performance. The GPU could be used as a computation model to extract its usefulness in terms of hiding the latency as it gives a large throughput due to presence of large number of computational units. GPU assisted scheduling can again improve the performance further. The proposed grouping of cores in this project could be implemented along with GPU assisted scheduling to obtain better performance.

1.1 Proposed Grouping Methods

Grouping could be done in two different perspective - either the core could restrict the processes to run on it or the process could itself restrict going on to a core. There are different ways of grouping cores based on different properties. In this project, I have proposed three different ways of restricting the processes to run on the predefined set of cores - which is essentially is the categorization of cores. One of the three, which was the best among them in terms of effectiveness, was chosen to implement and the performance was analysed.

1. Restricting the processes to run only on a predefined set of cores by blocking the migrations, when a process attempts to run on a core on which it is not supposed to run.
2. Preventing the process from entering in to the `runqueue` of a core on which it is not supposed to run.
3. A process could be restricted to run on a set of predefined cores alone which could be achieved by setting its allowed set of cores to run on.

1.2 Organization of The Thesis

Following this brief introduction, in Chapter - 2, I will brief on the three proposed methods for categorizing the groups and the possibilities for implementation. In Chapter - 3, implementation of the selected method out of the three will be discussed in detail. Chapter - 4 contains the experimentation results and I will conclude this report with Chapter - 5.

CHAPTER 2

METHODS FOR GROUPING CORES

I have proposed three different ways of grouping the cores here. Grouping of cores essentially means restricting the processes to run only on a predefined set of cores. For each of the methods different criteria are used. For each of the methods, its pros and cons are also discussed here.

2.1 By Blocking Process Migrations

In order to ensure an optimal performance, each and every process must be evenly treated considering it is priority values. When the scheduler schedules a process it takes several factors into consideration. If a process has been waiting in the queue for a long compared to others, it must be getting more priority over others and must be scheduled next on the available core. Scheduler will take care of all these scheduling related activities.

2.1.1 Process Migration

Active processes are placed in an array called the runqueue. The runqueue may contain priority values for each process, which will be used by the scheduler to determine which process to run next. To ensure each process has a fair share of resources, each one is run for some time period (quantum) before it is paused and placed back into the runqueue. When a program is stopped to let another run, the program with the highest priority in the runqueue is then allowed to execute. If any cores other than the currently allotted for a queued process is free to run a process, the process could be taken out from its current computing environment and put it on the other. That is the processes are moved from one computing environment to another which is referred to as Process Migration.

Process Migration could be either preemptive or non-preemptive. If the process migration takes place before starting the execution of the process, it is referred to as non-preemptive. While preemptive migration is where a process is preempted, migrated and continues processing in a different execution environment. The later type of migration is costly as it needs to recreate the process state.

Three criteria govern when a task can be migrated to another processor. First, the task is not running. Second, the destination processor is in the set of the allowed processors of the task. Third, the task is not cache hot on its current processor.

2.1.2 Blocking Process Migration

If we could prevent certain types of processes from getting migrated to a set of cores, in effect we can categorize the cores. For example, in an 8 core system, we wish to dedicate the 4 cores for processing only real-time tasks and rest 4 cores for other tasks. Such a categorization of cores could be achieved by blocking the migrations of real-time process from its runqueue to the others' runqueue.

We would have a desired predefined set of cores on which only a particular types of processes will be running. When the kernel decides to perform the migrations in order to ensure that each cores are getting processes evenly, as a pre-check we will be done to check whether the task is allowed to run on target core. If it is not, the migration will be blocked and it will continue until a core to which it could get migrated. Any process parameter could be used for the purpose of comparing it. If create a per-CPU variable, each processor on the system gets its own copy of that variable. It could be used to identify the allowed set of cores in this case.

2.1.3 Pros and Cons

- Since the blocking is happening only after the initialisation of process migration, it might cause unnecessary computations and thus leading to waste of time.
- Though it is an easy method to implement, it is not a very generic solution.
- It is difficult to dynamically set restrictions using per-CPU variables from the user space once the kernel is booted up.

2.2 By Restricting Entry to Runqueue

This method could be counted as a similar alternative to the previous one, Grouping by Blocking Migrations. The scheduler maintains a runqueue of all of the threads that are ready to be dispatched. When a more prior process is to be scheduled, it will get preempted and will be put on to the next available core. Unlike in the previous method, when the process is about to be put in to the runqueue itself, it could be checked whether the process is allowed to run on the particular core.

2.2.1 Restricting Entry to the Runqueue

The central data structure of the core scheduler that is used to manage active processes is known as the runqueue. Each CPU has its own runqueue, and each active process appears on just one runqueue. It is not possible to run a process on several CPUs at the same time. The scheduler puts the processes into the runqueue when it is in the active state, meaning when it is ready to get processed. At this stage, scheduler can check if the process type and whether it is allowed to run on the particular core or not. If it is not allowed to run on it, the scheduler can reject the process and assign it to the next scheduling class.

2.2.2 Pros and Cons

- This method is quite easy to implement by having a cross-checking when the kernel attempts to put a process in to the runqueue.
- Similar to previous method, it will not alter any properties or parameters associated with the task, that is the `task_struct`.
- This method will be fast enough but it might cause unnecessary computations.
- Not a generic solution. It is difficult to dynamically set restrictions using per-CPU variables from the user space once the kernel is booted up.
- Dynamic priority of the process might get changed when it is in the runqueue. So it is not a wise idea to choose this method as we might have to group the cores based on priority values of the processes also.

2.3 By Manipulating the Affinity Mask

Processor Affinity enables the binding and unbinding of a process or a thread to the core so that the process or thread will execute only on the designated core than any core. The concept of affinity could be used to restrict the processes from getting scheduled on a predefined set of cores in a system.

2.3.1 Affinity Mask

An affinity mask is a bit mask indicating what processors a thread should be run on by the scheduler of an operating system. By default at the forking stage each process will be set with a bit mask of all unity. Which essentially means, a process could be run on any available core.

2.3.2 Changing Affinity Mask

Affinity mask of a process could be changed at different stages during the life cycle of a process. It could be at forking stage when the process is being created or at pre-scheduling stage when the process is about to get scheduled. If a program in the user space wanted it to be run on a specific set of cores, it can achieve it by changing the mask via system calls. On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits.

For example, we can dedicate one core to a particular thread by setting the affinity mask of that thread to specify a single core, and setting the affinity mask of all other threads to exclude that core. It will help us to ensure maximum execution speed for that particular thread. But, usually the masks are not manipulated for any reason by default.

2.3.3 Pros and Cons

- This method is quite complex to implement, as the cpumask might get changed at different stages of the process life cycle.
- It will change the parameters in the `task_struct`. of the process. It might require more computations.

- Since the process parameters itself is changed and no more comparison or anything is required at later point of time during the process life cycle, it avoids unnecessary computations.
- It is generic solution, as we are not changing the parameters bound to the cores. Only process specific parameters are changed.

2.3.4 Advantages Over Previous Methods

In both the methods discussed previously, any process specific parameters are not being altered permanently. It just performs a few comparisons when ever required. While in this method, we would be changing the parameters associated with a process. Though it might take a few more computations compared to other methods, unnecessary ones are avoided. Since we are restricting the processes from accessing the cores via process parameters, applications can easily modify them from user space through kernel modules. Where in the previous methods, the core had to ensure that a not-allowed process is not getting scheduled on it.

In certain cases one will have to consider the priority of the tasks while grouping the cores. For example when we would like to dedicate a set of cores exclusively for real-time processes. The priority of the process could get changed anytime during the scheduling period or even after that. The previous methods for grouping the cores will fail in that case as it is not changing the process parameters dynamically. But the later method will update the mask in such situations and will succeed in classifying the cores correctly.

This method was counted as more advantageous methods over the other two. Implementation and the performance analysis of this method is discussed in subsequent chapters.

CHAPTER 3

IMPLEMENTATION

3.1 Introduction

The concept of Affinity Mask was manipulated to restrict the process from accessing a predefined set of cores. Based on the criteria for grouping, the Affinity Mask of the applicable processes could be changed in appropriate locations and thus it could be ensured that the processes are being scheduled and run only on the desired set of cores. The challenge is in setting the Affinity Mask (`cpumask`) where ever applicable. It has to be also ensured that the change is not affecting the performance hugely. The main aim is to come up with an optimal grouping strategy with lesser number of computations and improved performance.

3.2 Calculating the Affinity Mask

A CPU affinity mask is represented by the `cpu_set_t` structure (called as 'CPU Set') pointed to by `mask`. The `cpu_set_t` data structure represents a set of cores. CPU sets are used by `sched_setaffinity` and similar interfaces. The data structure is implemented as a bitmap - it states on which all cores a particular process is allowed to run. The generic structure of bit map (see Listing - 1) is defined in include/linux/-types.h, which is nothing more than an array of unsigned longs with a user-supplied name and length.

Listing 1: Generic Structure of bitmap

```
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
}
```

Manipulation of the masks could be done by the following predefined macros, in Table - 4.1. In our case, we have will have to define a function using these macros which will return the desired `cpumask`. We can also use macros like `CPU_AND()`, `CPU_OR()`, `CPU_XOR()` and `CPU_EQUAL()` to perform logical operations on the CPU sets.

`CPU_ZERO()`	Clears `cpuset`, so that it contains no CPUs.
`CPU_SET()`	Add CPU `cpu` to `cpuset`.
`CPU_CLR()`	Remove CPU `cpu` from `cpuset`.
`CPU_ISSET()`	Test to see if CPU `cpu` is a member of `cpuset`.
`CPU_COUNT()`	Return the number of CPUs in `cpuset`.

Table 3.1: Affinity Mask Macros

For example, if we wish to restrict a particular set of processes to run only on first four cores out of 8 available cores, firstly we have to calculate the `cpumask` using the above mentioned macros and then reset their masks. Bit map for the same would be `11110000` and its corresponding hexadecimal string would be `F0`.

Once we get the `cpumask` calculated using the macros or directly by setting the bit map, the mask has to be assigned to the process. The `sched_setaffinity()` can help us there.

3.3 Setting the Affinity Mask

In order to set the mask to a process we need to get the pointer to the corresponding `task_struct` and the desired mask to be set. The `sched_setaffinity()` function sets the CPU affinity mask of the thread whose ID is `pid` (which is available from `task_struct`) to the value specified by the `mask`. In case the `pid` is zero, then the calling thread will be used. If the thread specified by `pid` is not currently running on one of the CPUs specified in `mask`, then that thread is migrated to one of the CPUs specified in `mask`, by default. The affinity mask is a per-thread attribute that can be adjusted independently for each of the threads in a thread group. After a call to `sched_setaffinity()`, the set of CPUs on which the thread will actually run is

the intersection of the set specified in the mask argument and the set of CPUs actually present on the system. Once the mask is set with the desire one, the restrictions will come by default. The scheduler will consider the mask while scheduling the processes. The core itself does not have to worry about restricting certain processes from running on it. The affinity mask will take care of it.

Task Structure

The processes in Linux are a group of threads and the kernel schedules the threads, not the processes. The term `task` is generally used to represent a Thread. The data structure that contains all the information about a specific task is `task_struct` which is defined in include/linux/shed.h header file. This structure contains all of the necessary data to represent the process, along with a plethora of other data for accounting and to maintain relationships with other processes (parents and children).

Refer Listing - 2, which shows some of the important parameters in `task_struct` which will come in to my discussion in subsequent sections.
The `state` variable is a set of bits that indicate the state of the task. The following are the most common states. There are a few more different states mentioned in ./linux/include/linux/sched.h.

- `TASK_RUNNING` - Process is running or in a run queue about to be running.
- `TASK_INTERRUPTIBLE` - If the process is sleeping.
- `TASK_UNINTERRUPTIBLE` - Process is sleeping but unable to be woken up.
- `TASK_STOPPED` - If the process is stopped.

Each process is also given a priority (called `static_prio`), but the actual priority of the process is determined dynamically based on loading and other factors. The lower the priority value, the higher its actual priority.

The `tasks` field provides the linked-list capability. It contains a `prev` pointer (pointing to the previous task) and a `next` pointer (pointing to the next task). The `pid` refers to the unique ID of the process. Other parameters shown in the Listing - 2 is out of scope in our discussion.

Listing 2: A small portion from task_struct

```
struct task_struct {

        volatile long state;
        void *stack;
        unsigned int flags;

        int prio, static_prio;

        struct list_head tasks;
        struct mm_struct *mm, *active_mm;

        pid_t pid;
        pid_t tgid;

        struct task_struct *real_parent;
        char comm[TASK_COMM_LEN];
        struct thread_struct thread;
        struct files_struct *files;
        ...
};
```

Accessing task_struct Parameters

In most cases, processes are dynamically created and represented by a dynamically allocated `task_struct`. One exception is the `init` process itself, which always exists and is represented by a statically allocated `task_struct`. When a process is created by `fork()` system call, the whole content of the parent `task_struct` will be copied to the child. All the processes in Linux is collected in to a task list - but it is not accessible from the userspace. For our purpose, we need it to be made accessible from the userspace. A small code was inserted into the kernel in the form of a module in order to retrieve the information about each task. The soul part of the kernel module is shown in Listing 3.

Listing 3: Kernel Module to fetch task information to the userspace

```
...
/* Initialize a point to halt the iteration */
struct task_struct *task = &init_task;

/* Iterate through the linked list of tasks
   until it finds the init_task again */
do {

  printk(KERN_INFO "%s[%d]_parent_%s\n",
   task->comm, task->pid, task->parent->comm);

} while ((task = next_task(task)) != &init_task);
...
```

3.3.1 Changing Mask at Process Creation Stage

A new process is created from the userspace essentially via `do_fork()`. Even for the creation of kernel threads, firstly `kernel_thread()` is called which in turn calls `do_fork()` after making some initialisations. It has another partner function which is `copy_process()`. The `task_struct` of the parent is 'blindly' copied to the child. The `copy_process` function, called by `do_fork` is where the new process is created as a copy of the parent. The `copy_process` function calls `dup_task_struct` function which allocates a new `task_struct` and copies the current process's descriptors into it.

At this stage we could manipulate the affinity mask `cpus_allowed`, with the desired mask. It is not compulsory to change it here. But there is a narrow possibility that the newly created process might get scheduled immediately. It will save such cases. Later the `do_fork` function calls to `wake_up_new_task` to make the new task 'running'.

3.3.2 Changing Mask at Scheduling Stage

About the Scheduler

The default scheduler is Completely Fair Scheduler (CFS). CFS ensures that each process is getting fair amount of the processor. CFS maintains a time-ordered red-black tree rather than the tasks in a run queue, which has been done in prior Linux schedulers. With tasks (represented by `sched_entity` objects) stored in the time-ordered red-black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree. The scheduler chooses the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks.

The generic `schedule()` function, which preempts the currently running task. The currently running preempted task is returned to the red-black tree through a call to `put_prev_task` through the scheduling class. The `schedule` functions picks up the next task to schedule by calling `pick_next_task` function. It will just pick up the left-most task from the red-black tree and returns the associated `sched_entity`, which includes the `rb_node` reference, load weight, and a variety of statistics data. Calling `task_of()` will identify the `task_struct` and the generic scheduler finally provides a processor to this task.

Setting the Mask

Each task belongs to a scheduling class, `sched_class`, which defines a set of functions that decides the behavior of the scheduler - which in turn determines how a task will be scheduled. The `schedule()` function internally calls the `__setscheduler` function (shown in Listing - 4) which essentially assigns the scheduler class to a task. This is where the priority value of the task is compared and put into the corresponding scheduler class. Since this is the entry point of a task to the scheduler, the affinity mask must be changed with the desired mask. We can define a function which could be called from the `__setscheduler` function which will fetch the desired mask and replace the default mask.

Listing 4: Part of schedule() function

```
static void __setscheduler (struct rq *rq,
struct task_struct *p, const struct sched_attr *attr){
        __setscheduler_params(p, attr);
        p->prio = normal_prio(p);
        if (dl_prio(p->prio))
                p->sched_class = &dl_sched_class;
        else if (rt_prio(p->prio))
                p->sched_class = &rt_sched_class;
        else
                p->sched_class = &fair_sched_class;
}
```

3.4 Dedicating Cores for Real-Time Processes

The grouping of cores was tested by dedicating a set of cores to perform only the real-time processes. The aim was to group all the available 32 cores in the laboratory machine, in certain ratio such that a set of cores are dedicated for running the real-time tasks alone. This section explains only the way it was achieved.

The suggested way for implementing it was as follows. Define a function, say name it `set_cpumask()`, which will give you the desired mask to be set checking its priority value. Say we wish to dedicate first 16 cores exclusively for real-time processes and next 16 for other processes out of the total 32 available cores. Let the `set_cpumask` function take the `task_struct` as its input. Based on the priority value in the `task_struct`, if it is in real-time range (that is 0-99 range), set the mask as `FFFF0000`.

The mask could be changed at the process is creation stage and when it gets scheduled by the scheduler. During process creation, when the `do_fork` function is called, call the `set_cpumask` function also. Do the same when the scheduler class is getting assigned for a process. It could be thus ensured that only the real-time processes are getting scheduled on the first 16 set of cores out of available 32.

This proposed method was implemented in BOSS MOOL along with another project titled *Grouping of Cores and Loadbalancing in BOSS MOOL*. Experimentation was done by me on it to analyse its performance. Results of the same could be found in the next chapter.

CHAPTER 4

EXPERIMENTATION AND RESULTS

Implementation of the method, discussed in Section - 3.4 was experimented with the several cases to analyse the performance. The basic aim was to find out the factors affecting the performance in grouping of cores and how grouping could be made more optimal. The experimentation was performed on the laboratory machine which has 32 cores. Benchmarking tools used for this purpose were sysbench and lmbench.

Experiment - 1

Cores were grouped in a ratio 1:1 dedicating half of the available cores for real-time processes alone. Cores were ordered in XXX...YYY... fashion, that is the cores dedicated for processing real-time tasks were grouped together. The table 4.1 shows the time taken to run different loads in such a grouping of cores, using different benchmark tools. Load is in 'No. of Threads' and the time recorded is in 'Milli Seconds (ms)'.

Benchmark Tool - sysbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	7471.65	7402.73	7396.08	7434.62	7216.44	7384.30
5000	3687.71	3900.53	4172.12	4415.73	4015.73	4038.36
1000	929.44	993.18	996.58	1014.09	1001.51	986.96
500	463.18	477.55	465.09	470.35	454.41	466.12

Benchmark Tool - lmbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	6982.23	7002.36	6985.23	6873.94	7056.41	6980.034
5000	3687.71	3900.53	4172.12	4415.73	4015.73	4038.364
1000	930.26	958.12	971.46	912.78	936.49	941.822
500	426.98	436.71	430.12	428.78	411.51	426.82

Table 4.1: Grouping of cores in 1:1 ratio - Grouped

Observation

It was observed using `htop` that half the cores were dedicated only for the real-time tasks and remaining for the rest. The figure - 4.1 is a screenshot of Terminal window with `htop` running while the benchmark tool was running in background. It can be seen from the figure that only half the cores are getting assigned with processes as the benchmark tool is not creating any real-time threads at the given instance.

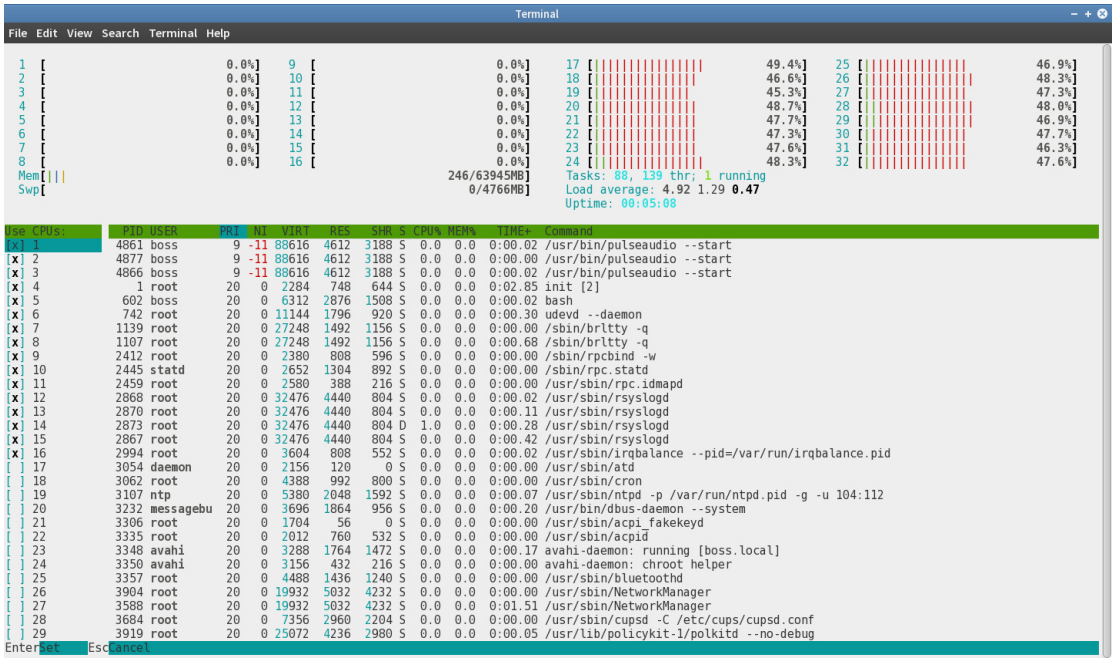

Figure 4.1: Screenshot of Terminal with htop running

Experiment - 2

Cores were grouped in a ratio 1:1 dedicating half of the available cores for real-time processes alone. Cores were ordered in X-Y-X-Y-X-Y... fashion, that is alternatively. The table 4.2 shows the time taken to run different loads in such a grouping of cores, using different benchmark tools. Load is in 'No. of Threads' and the time recorded is in 'Milli Seconds (ms)'.

Benchmark Tool - sysbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	11797.78	11802.58	11690.77	11803.25	11723.28	11763.53
5000	6376.01	6377.33	6370.92	6332.87	6369.18	6365.26
1000	1037.17	1045.88	1035.75	1049.75	1023.83	1038.48
500	471.53	481.61	487.71	490.41	503.27	486.91

Benchmark Tool - lmbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	10236.25	10892.12	11002.31	10521.26	10072.13	10544.814
5000	5892.13	5761.26	5396.21	5846.12	5916.38	5762.42
1000	1021.84	1010.89	896.36	956.12	989.76	974.994
500	426.38	419.46	456.28	436.78	503.27	448.434

Table 4.2: Grouping of cores in 1:1 ratio - Alternatively

Experiment - 3

Cores were grouped in a ratio 1:1 dedicating half of the available cores for real-time processes alone. Cores were ordered in XX-YY-XX-YY-... fashion, that is double alternatively. The table 4.3 shows the time taken to run different loads in such a grouping of cores, using different benchmark tools. Load is in 'No. of Threads' and the time recorded is in 'Milli Seconds (ms)'.

Benchmark Tool - sysbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	10598.86	10289.76	10613.61	10914.41	10481.73	10579.67
5000	5895.24	5855.49	5785.71	5782.63	5689.32	5801.68
1000	1480.26	1517.68	1465.81	1506.37	1454.23	1484.87
500	722.63	721.29	700.38	686.27	769.61	720.04

Benchmark Tool - lmbench						
Load	Test-1	Test-2	Test-3	Test-4	Test-5	Average
10000	10023.58	10096.36	10189.65	10127.86	10097.92	10107.074
5000	5623.78	5469.35	5583.47	5632.15	5557.12	5573.174
1000	1396.56	1375.81	1412.01	1456.48	1386.57	1405.486
500	683.46	693.45	675.23	705.26	698.73	691.226

Table 4.3: Grouping of cores in 1:1 ratio - Double Alternatively

Comparison of Different Groupings

The table 4.4 has the data from two different benchmark tools, sysbench and lmbench. The average time (in 'ms') taken for the processing of corresponding load (in 'No. of Threads') is recorded in the table.

Benchmark Tool - sysbench				
Load	**10000**	**5000**	**1000**	**500**
Alternate	11763.53	6365.26	1038.48	486.91
Double Alternate	10579.67	5801.68	1484.87	720.04
Grouped	7384.30	4038.36	986.96	466.12

Benchmark Tool - lmbench				
Load	**10000**	**5000**	**1000**	**500**
Alternate	10544.81	5762.42	974.99	448.43
Double Alternate	10107.07	5573.17	928.61	430.29
Grouped	8780.03	5338.36	941.82	426.82

Table 4.4: Comparison of different grouping

The figures - 4.2 and 4.3 are the graphs (Load-vs-Time) plotted using the data obtained from different grouping methods - Alternate, Double Alternate and Grouped. First graph uses the data from sysbench tool and the later one uses data from lmbench tool.

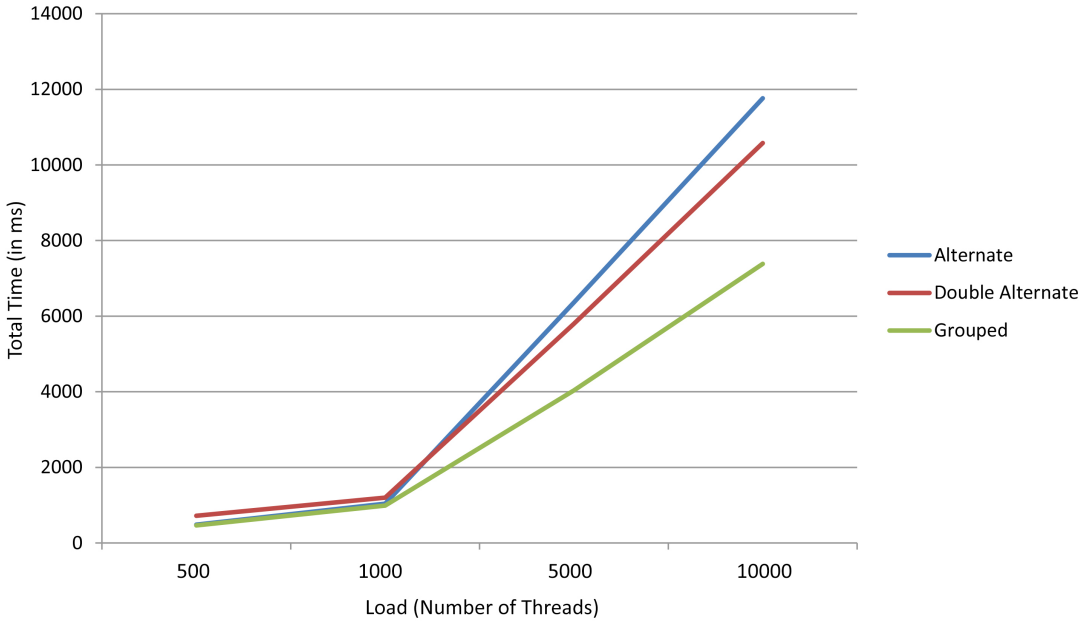

Figure 4.2: Comparison of different grouping methods (using sysbench data)

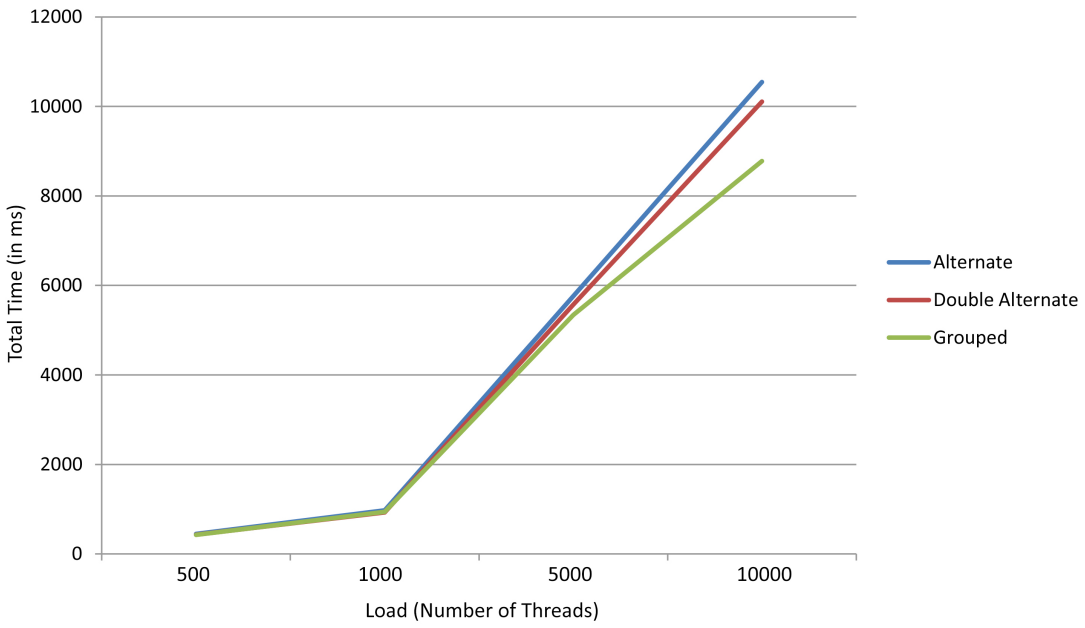

Figure 4.3: Comparison of different grouping methods (using lmbench data)

Observation

For relatively huge workloads, it is observed that it takes lesser time for processing when the cores are grouped together (as XXX...YYY..) compared to the alternate grouping (as X-Y-X-Y-X-Y...) and the double alternate grouping (as XX-YY-XX-YY-...).

Experiment - 4

The ratio in which the cores were grouped, was altered to 2:1 (ie., two-third of the total available cores were dedicated for real-time tasks and the remaining for other tasks), 1:2, 3:1 and 1:3 - for the purpose of experimentation. Since the threads created by the benchmark tools were not distinguishable as real-time or the other, it was not useful. The results of the Experiment - 4 could not conclude anything substantially. It could be inferred that, the ratio in which the cores are to be grouped must be based on the nature of the workload. Say for example, if the major percentage of the workload is real-time tasks, more number of cores must be allocated to it, in order to achive optimal performance.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Grouping the available number of cores in a multi-core system and dedicating each set of cores to perform a certain type of tasks would improve the overall performance. There are different possible ways for grouping the cores from two different perspectives. First one from the cores' perspective where the cores could block certain type of processes from accessing it. Second one from the processes' perspective where the process itself could restrict is getting scheduled on a predefined set of cores.

The later one takes lesser number of computations and avoids unnecessary ones, and thus more optimal compared to other methods. Being a more optimal and generic solution, it is concluded to be the more effective way of grouping the cores. Irrespective of the ratio in which the cores are grouped, better performance is obtained when the cores are grouped together. The alternate arrangement of cores takes more time especially when the workload is huge.

5.2 Future Work

The criteria for classifying the cores could be different. It is difficult to generalize an optimal ratio in which the cores could be grouped. The ratio could be calculated dynamically at runtime as it is workload dependent. Finding a generalized solution to calculate the optimal ratio dynamically would improve the performance further.

REFERENCES

[1] **W. Mauerer** *Professional Linux Kernel Architecture* (2008). Wrox Press Ltd., Birmingham, UK. ISBN 0470343435.

[2] **Bovet, Daniel P. and Marco Cesati** *Understanding the Linux Kernel* (2005). O'Reilly Media, Inc.

[3] **Pabla, Chandandeep Singh** *Completely Fair Scheduler* (2009). Linux Journal 2009.184 (2009): 4

[4] **Janakiram Dharanipragada, Hemang Mehta, and S. J. Balaji** *Dhara: A Service Abstraction-Based OS Kernel Design Model* (2012). Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on. IEEE

[5] **M. Tim Jones** *Anatomy of Linux Process Management* (2008). http://www.ibm.com/developerworks/library/l-linux-process-management [URL]

[6] **Volker Seeker** *Process Scheduling in Linux* (2013). CriticalBlue - The University of Edinburgh. http://criticalblue.com/uploads/2013/12/linux-scheduler.pdf [URL]

[7] **Bowden T. and B. Baue** *The proc File System* (1999, 2009 - recent update). https://www.kernel.org/doc/Documentation/filesystems/proc.txt [URL]

[8] *Linux Kernel Documentation*. https://www.kernel.org/doc [URL]