RATS: Resource Aware Thread Scheduling for JVM level Clustering

Navaneeth Rameshan
navaneeth.rameshan@gmail.com

ABSTRACT

In this work, we propose RATS, a middleware to enhance and extend the Terracotta framework for Java with the ability to transparently execute multi-threaded Java applications to provide a single-system image. It supports efficient scheduling of threads, according to available resources, across several nodes in a Terracotta cluster, taking advantage of the extra computational and memory resources available. It also supports profiling to gather application characteristics such as dispersion of thread workload, thread inter-arrival time and resource usage of the application. It uses byte-code instrumentations to profile and add clustering capabilities to multi-threaded Java applications, as well as extra synchronization if needed. We developed a range of alternative scheduling heuristics and classify them based on the application and cluster behavior. The middleware is tested with a cpu-intensive application with varying thread characteristics to assess and classify the scheduling heuristics with respect to application speed-ups and load balancing.

1. INTRODUCTION

If the workstations in a cluster can work collectively and provide the illusion of being a single workstation with more resources, then we would have what is referred in the literature as a Single System Image [3]. Much research has been done in the area of SSIs, such as Distributed Shared Memory (DSM) systems and Distributed Virtual Machines that can run applications written in a high-level language in a cluster, behaving as if it were on a single machine. One of the essential mechanisms necessary for providing SSI systems is the scheduling of threads for load balancing across the cluster. The current most popular system that uses a shared object space is Terracotta. At present it does not support scheduling of threads and instead multiple manual instances needs to be launched to scale applications. We propose RATS, a Resource Aware Thread Scheduling for JVM level Clustering which is an extension of Caft [8]. Caft provides full transparency for running multi-threaded applications. RATS bridges the gap between transparency and efficient scheduling of threads using Terracotta to keep data consistent across the cluster and scale existing applications with ease.

Several studies have showed that no single scheduling algorithm is efficient for all kinds of applications. RATS supports multiple scheduling heuristics and they behave differently for different characteristics of applications. These scheduling heuristics are classified based on the properties of the application and can be used efficiently to suite any class of application. RATS provides a profiler that allows to characterize an application based on the dispersion of thread workload, thread inter-arrival time and the resource usage of the application. The information obtained from the profiler allows to opt for the most efficient thread scheduling heuristic. The scheduling heuristics maintain state information of the worker in the form of resource usage and threads launched to make optimal decisions. RATS allows to run an already existing application in a distributed manner using a scheduling heuristic that best suites the characteristics of the application and the cluster.

The rest of the document is organized as follows. We provide a brief background of the most relevant related work followed by a description of the architecture and the supported scheduling heuristics along with the profiling abilities. Finally we evaluate the scheduling heuristics for different class of applications and cluster properties.

2. RELATED WORK

There are three major approaches that exist for distributed execution in a cluster. They are: Compiler-based Distributed Shared Memory systems, Cluster-aware Virtual Machines and systems using standard Virtual Machines. Compiler-based Distributed Shared Memory Systems (DSM) is a combination of a traditional compiler with a Distributed Shared Memory system. The compilation process inserts instructions to provide support for clustering without modifying the source code. Jackal [13] compiler generates an access check for every use of an object field or array element and the source is directly compiled to Intel x86 assembly instructions, giving the maximum performance of execution possible without a JIT. Jackal does not support thread migration.

Cluster-aware Virtual Machines are virtual machines built with clustering capabilities in order to provide a Single System Image (SSI). cJVM[2] is able to distribute the threads in an application along with the objects without modifying the source or byte code of an application. It also supports thread migration. To synchronize the objects across the cluster a master copy is maintained and updated upon every access and is a major bottleneck. In Kaffemik [1], all objects are allocated in the same virtual memory address across the cluster thus allowing a unique reference valid in every instance of the nodes. However, it does not support caching or replication and can result in multiple memory accesses, thus reducing performance.

Systems using Standard VMs are built on top of a DSM system to provide a Single System Image for applications. Some of the most popular systems are java party [14], java Symphony[5] and JOrchestra[12]. J-Orchestra uses bytecode transformation to replace local method calls for remote method calls and the object references are replaced by proxy references. Java Symphony allows the programmer to explicitly control the locality of data and load balancing. All the objects needs to be created and freed explicitly which defeats the advantage of a built-in garbage collection in JVM. Java party allows to distinguish invocations as remote and local by modifying the argument passing conventions. The implementation does not satisfy the ideal SSI model as classes need to

be clustered explicitly by the programmer.

Some of the classic scheduling algorithms that are most relevant to thread scheduling is explained in the following. In First Come First Served algorithm, execution of jobs happen in the order they arrive ie. the job that arrives first is executed first [7]. If a large job arrives early, all the other jobs arriving later are stalled in the waiting queue until the large job completes execution. This affects the response time and throughput considerably. This disadvantage is overcome by Round Robin. In this algorithm algorithm every job is assigned a time interval, called quantum, during which it is allowed to run [11]. Since jobs execute only for a specified quantum, the problem of larger jobs stalling jobs that arrive later is mitigated. The Minimum Execution Time (MET) algorithm assigns each task to the resource that performs it with the minimum execution time [9]. MET does not consider whether the resource is available or not at the time (ready time) [4, 9, 10] and can cause severe imbalance in load across resources [4, 9, 10]. The Min-min algorithm has two phases [4]. In the first phase, the minimum completion time of all the unassigned tasks are calculated [10]. In the second phase, the task with the minimum completion time among the minimum completion time that was calculated in the first phase is chosen. It is then removed from the task list and assigned to the corresponding resource [4]. The process is repeated until all the tasks are mapped to a resource. In the Suffrage algorithm the criteria to assign a task to a resource is the following: assign a resource to a task that would suffer the most if that resource was not assigned to it [7, 9]. In order to measure the suffrage, the suffrage of a task is defined as the difference between its second minimum completion time and its minimum completion time [6, 10]. These completion times are calculated considering all the resources [7]. Once a task is assigned to a resource it is removed from the list of unassigned tasks and the process is repeated until there are no tasks in the unassigned list.

3. ARCHITECTURE

This section describes the architecture of the middleware, implemented to allow Terracotta to schedule threads for simple multi-threaded java applications on a cluster. RATS middleware consists of two components - A Master and Worker. The master is responsible for running the application and launches threads remotely on the worker nodes. The worker exposes an interface for launching threads and provides all the operations supported by java.Lang.Thread class. The master, on the other hand is responsible for launching the application with an executable jar and uses a custom class loader that loads the class after performing necessary instrumentation to the application code. Figure 1 provides a high level view of the RATS architecture.

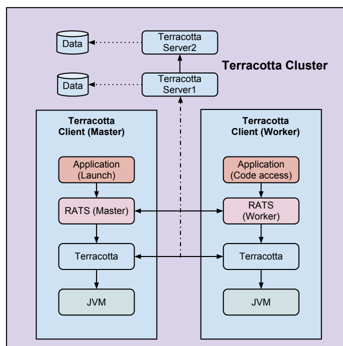

Figure 1: Architecture of RATS

RATS was implemented by modifying an existing middleware called CAFT (Cluster Abstraction for Terracotta) [8]. CAFT provides basic support for remote thread spawning along with transparency. RATS extends CAFT to enhance byte code instrumentation along with support for multiple resource aware scheduling algortihms. To understand how the master worker paradigm allows for remotely spawning threads, we first provide a high level architecture of the communication between master and worker and in the following subsection we explain the different scheduling heuristics the system supports.

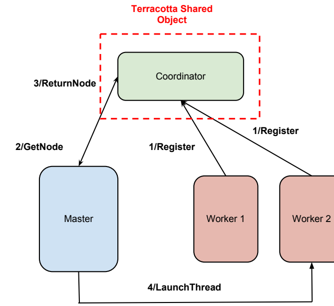

Figure 2: Master-Worker Communication

Figure 2 shows the communication between different components that are required to remotely spawn a thread. As soon as the workers are launched, they first register themselves with the coordinator (1/Register). The coordinator acts as an interface between the worker and master and is used for sharing information between each other. When the application is launched, and a thread is instantiated, the master communicates with the coordinator component to fetch the node for launching the thread (2/GetNode). The coordinator communicates with other components responsible for scheduling and returns the node (3/ReturnNode). Upon receiving the information of the node for remotely spawning the thread, the master finally launches the thread on the worker (4/LaunchThread). Here in this example, worker 2 is chosen for running the thread.

Scheduling Heuristics. When an application launches a thread, the master is responsible for making scheduling decisions based on the chosen heuristic. The worker can also make scheduling decisions if a thread spawns multiple threads. The middleware supports two types of scheduling and they are presented below:

Centralized Scheduling. In centralized scheduling, the decisions are taken entirely by a single node. Here, the master is responsible for making every scheduling decision. Based on the specified heuristic, the master selects a worker for remotely executing the thread and also maintains state information. The centralized scheduling heuristics supported by the middleware are:

Round-Robin : In round-robin scheduling, the threads launched by the application are remotely spawned on the workers in a circular manner. Threads are launched as and when they arrive and the scheduling is static by nature. It does not take into account any information from the system and the workers are chosen in the order they registered with the master.

Resource-Load: Scheduling decisions are made depending on the load of every worker. The supported scheduling algorithms based on load information are:

- **CPU-Load**: The CPU load of every worker is monitored by the master and the threads are remotely launched on the worker with the least CPU load. The master maintains state

information about the CPU load of every worker. The load information of CPU and memory of the system is obtained using the library SIGAR.

- **Load-Average**: Threads are scheduled on nodes with the least cpu utilization until the cpu load gets saturated. The scheduling heuristics then aims at equalizing the load average across the cluster. The information about load averages in linux are obtained from the command line utility top. However, the values of load average obtained from top are not instantaneous. They are measured in three ranges as a moving average over one minute, five minute and fifteen minutes. In all Linux kernels the time taken for updating the moving average is five seconds. If multiple threads are launched instantaneously within a five second window, it is possible that all the threads are launched on the worker with the lowest load average. This problem is circumvented by an estimation of number of threads to launch based on the current values of load average.

Listing 1: Scheduling heuristic based on load average

```
if (loadAvgMonitor == true){
        for each worker:
                if(avgLoad< NumberProcessors)
                        avgLoadMap.put(nodeID, NumberProcessors —
                            avgLoad)
                else
                        avgLoadMap.put(nodeID,1)
        loadAvgMonitor=false
}
selectedNode = NodeID with maximum value in avgLoadMap
avgLoadMap.put(selectedNode, value—1)
if (all values in avgLoadMap.valueSet == 0){
        loadAvgMonitor = true
}
```

In the pseudocode listed in listing 1, the number of threads to be scheduled on a worker is inversely proportional to the load average. If the load average is less than the number of processors, only so many threads are launched to fill up the processor queue to the number of processors in the worker. Any further load monitoring is performed only after all these threads are scheduled.

- **Accelerated-Load-Average**: This heuristic is similar to the scheduling heuristic Load-Average but is not as conservative and takes into account instantaneous changes in load average. It allows for scheduling the minimum number of threads possible while keeping the estimation correct and at the same time aiding in using a recent value of load average. Similarly, difference in load average is also inversely proportional to the number of threads to be scheduled. In order to achieve scheduling of at least one thread on any worker, the highest difference in load average is remapped to one and other differences are shifted accordingly.

Listing 2: Scheduling heuristic based on accelerated load average

```

if (loadAvgMonitor == true){
        if(!first run){
        for each worker:
                avgLoadDiffMap.put(nodeID, AvgLoadMap — prevAvgLoadMap)
        }
        if (first run || all values in avgLoadDiffMap.valueSet == 0){
                for each worker:
                        if(avgLoad< NumberProcessors)
                                avgLoadMap.put(nodeID, NumberProcessors
                                    — avgLoad)
                        else
                                avgLoadMap.put(nodeID,1)
                loadAvgMonitor=false
        }
        //remapping maximum value in loadAvgDiff to 1
        if(all values in avgLoadDiffMap.valueSet != 0){
                for each worker:
                        avgLoadMap.put(nodeID, loadAvgDiff.currentvalue/
                            max(loadAvgDiff.valueSet))
        }
        Copy values avgLoad to prevAvgLoadMap
        // remapping minimum value in avgLoadMap to 1
        for each worker:
                avgLoadMap.put(nodeID, avgLoadMap.currentvalue/ min(
                    avgLoadMap.valueSet))
}
selectedNode = NodeID with maximum value in avgLoadMap
avgLoadMap.put(selectedNode, value—1)
if (all values in avgLoadMap.valueSet == 0){
        loadAvgMonitor = true
}
```

The load information of CPU and load average is updated by the worker in one of the two ways:

1. *On-demand*: When an application is just about to launch a thread, the master requests all the workers to provide their current CPU load. State information is updated only on demand from the master. This is a blocking update and it incurs an additional overhead of round trip time delay to every worker for every thread launch.
2. *Periodic*: The load information of CPU maintained by the master is updated after a constant period. The period required to perform updates is a configurable parameter which can be chosen by the user. All updates are performed asynchronously and hence they do not block remote launching of threads.

- **Thread load**: The master maintains state information about the number of threads each worker is currently running. The scheduling heuristic makes decisions to launch threads on workers with the least number of currently executing threads. This heuristic schedules in a circular fashion just like round robin until at least one thread exits. Once a thread exits, it ceases to behave like round robin. The state information is updated only when a thread begins or finishes execution.

Hybrid Scheduling. Once a thread is scheduled to a worker, depending on the application, the thread itself may launch more internal threads. To handle such scenarios, the Middleware also supports hybrid scheduling, where local copies of information that help scheduling decisions are maintained. The trade-off between consistency and performance is handled optimally for distributed scheduling.

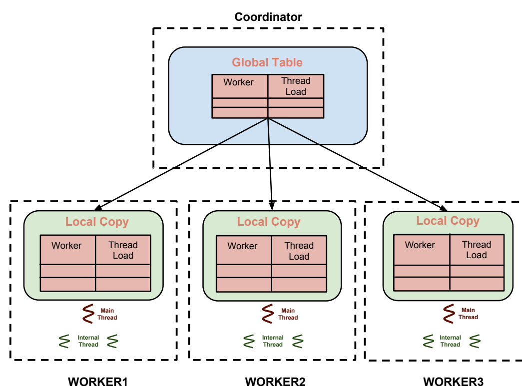

Figure 3: Communication for Worker performing Scheduling from Local Information

In this approach, the master asynchronously sends the state information table to every worker before any thread launch. The

workers on receiving the information table store a copy of the table locally. This is shown in figure 3. Workers use this local table for making scheduling decisions after which they update the local table and then the global table. Once a worker updates its local table, there are inconsistencies between the information table between the workers. Although there are inconsistencies, they are lazily consistent and the final update on the global table is always the most recent and updated value. We achieve this by considering updates only to entries corresponding to that worker, in both the global and the local table. This restriction prevents updates to global table from blocking.

In this context, performance and consistency are inversely proportional to each other and we aim to improve performance by sacrificing a bit on consistency. If a worker has to schedule based on thread load and makes a choice by always selecting the worker with the least loaded node from its local table, then it could result in every worker selecting the same node for remotely spawning an internal thread, eventually overloading the selected node. This happens because the workers do not have a consistent view of the information table for a certain period. To prevent this problem, workers make their choice based on weighted random distribution.

Profiling. The middleware allows for profiling an application in order to choose the best scheduling heuristic for efficient load balancing and performance. In this section we discuss the metrics measured by the profiler and present an optimal period for updating the state information at the master. The metrics measured by the profiler are dispersion of thread work load, thread inter arrival times and resource usage of the application. These metrics help the user choose the right scheduling heuristic to gain maximum performance. By measuring these metrics it is possible to choose an optimal period for periodic updates of worker load information to the master. Due to space constraint the derivation of the optimal period is omitted. Optimal period for a worker is given by:

$$p = \sqrt{\frac{t_l * (2 * t_m + RTT)}{2 * N}}$$

where t_l is the arrival time of the last thread, t_m is the time to monitor load by the worker, RTT is the round trip time to the Terracotta server and N is the total number of threads.

4. EVALUATION

We used up to three machines in a cluster, with Intel(R) Core(TM)2 Quad processors (with four cores each) and 8GB of RAM, running Linux Ubuntu 9.04, with Java version 1.6, Terracotta Open Source edition, version 3.3.0, and a cpu-intensive multi-threaded Java applications that has the potential to scale well with multiple processors, taking advantage of the extra resources available in terms of computational power.

Overhead Incurred. Executing java applications on the middleware incurs an additional overhead of increase in the size of bytecode and delay in launching a thread. Depending on the scheduling algorithm used, there may be additional overhead in updating load information. Periodic updates of load information are asynchronous and it is difficult to directly measure the overhead incurred. For this reason and to maintain generality, this section measures the overhead involved in launching a thread and the increase in the size of bytecode.

It can be observed from Table 1 that the size of bytecodes have increased because of additional instrumentations to provision transparency, remote launching of threads, monitoring runnable objects and capturing thread execution times. This increase in the size of bytecode does not consider the instrumentations done by Terracotta. As the number of threads in the MD5 hashing application doubles, the total overhead incurred for launching threads also doubles. The overhead is considerable and indicates that the middleware is not suited for applications that are compute non intensive.

From Table 2, it can be seen that the percentage increase in the size of byte code added by the middleware is only 6.8%. However, the average time taken to launch a thread for fibonacci generation is different from the average time taken to launch a thread for MD5 hashing. Apart from scheduling decision and RTT, it also involves the time taken to store the runnable object in the virtual heap. As the size of the runnable object increases, the overhead also increases. The total time taken to launch threads doubles as the number of threads double. In order to achieve any gain in performance, the gain obtained must be greater than the overhead incurred. Otherwise, the middleware deteriorates the performance of the application. The middleware is thus suited for compute intensive applications.

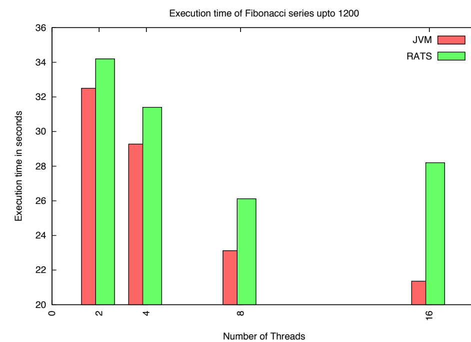

Figure 4: Execution time for Fibonacci number generation.

Execution Time. Fibonacci number generation is configured to compute the first 1200 numbers of the Fibonacci sequence, with number of threads directly proportional to the number of processors available. The time taken with two, four, eight and sixteen threads is measured. The application is tested in a standard local JVM, for comparison with the distributed solution. The distributed solution executes the application with two workers. Figure 4 compares the execution times for different number of threads.

It can be seen from the figure that the middleware decreases the performance of the application. Fibonacci number generation is not compute intensive, hence the overhead incurred in launching the threads and gathering the results, exceeds the gain obtained in execution time. As the number of threads double, the execution time taken using the middleware increases drastically. Since the application is evaluated for 1200 numbers, the load of the application remains a constant. This means that, increasing the number of threads decreases the load per thread. As the load per thread decreases, the gain obtained by distributed execution, decreases, but the overhead incurred in launching threads increases. Hence, distributed execution of the application deteriorates the performance, when the number of threads increase for a constant load. Fibonacci is highly recursive and it ends up allocating pages for stacks, which on a fast processor leaves a lot of available CPU for other threads. There is a lot of context switching for very small functions.

Bytecode Instrumentation Overhead		
Original size	**After Instrumentation**	**Percentage Increase**
3367 bytes	3539 bytes	5.04 %

Thread Launch overhead			
No. of threads	**Avg. time to launch a thread**	**Total overhead**	**Percentage Increase**
2	0.39 secs	0.78 secs	-
4	0.39 secs	1.58 secs	100.5 %
8	0.39 secs	3.16 secs	100 %
16	0.39 secs	6.27 secs	98.4 %

Table 1: Overhead for MD5 Hashing

Bytecode Instrumentation Overhead		
Original size	**After Instrumentation**	**Percentage Increase**
6275 bytes	6702 bytes	6.8 %

Thread Launch overhead			
No. of threads	**Avg. time to launch a thread**	**Total overhead**	**Percentage Increase**
2	0.52 secs	1.04 secs	-
4	0.51 secs	2.04 secs	96.15 %
8	0.51 secs	4.08 secs	100 %
16	0.51 secs	8.16 secs	100 %

Table 2: Overhead for Fibonacci generation

For measuring the performance of the web crawler, it was tested under different scenarios. The number of websites crawled were increased for each evaluation. Number of threads for crawling within a single website is maintained as a constant at three. These three threads require synchronization among themselves. Every website is crawled up to a depth of two. Execution time is measured for crawling ten, twenty and thirty websites. Since the difference in execution times vary extremely, results obtained for crawling ten websites is plotted separately and is shown in Figure 5.

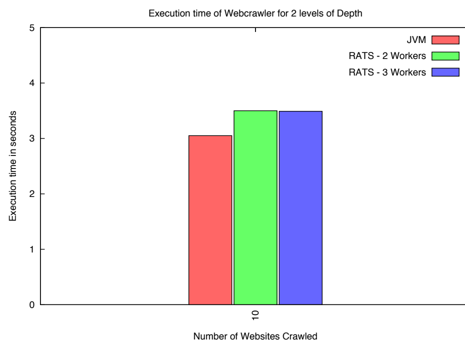

Figure 5: Execution time for web crawler - 10 websites.

As the number of threads increase within a single JVM, the thread idle time increases, because of contention for the available processors. Any gain is achieved by minimizing this idle time. By distributing the threads on multiple workers, the idle time is greatly reduced as the number of threads per processor decreases. From Figure 5, it can be seen that the time taken to crawl ten websites until depth two is only three seconds. The overhead incurred in launching ten threads alone exceeds three seconds. Thus, any gain obtained in minimising the idle time is not visible. But as the size

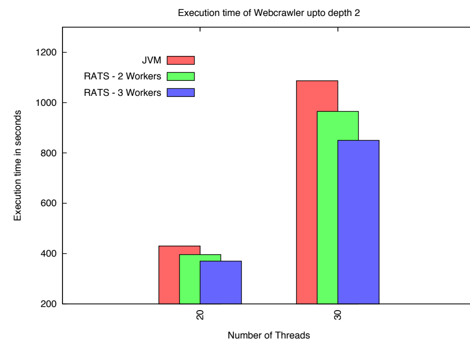

Figure 6: Execution time for web crawler - 20 and 30 websites.

and the number of websites increase, the time taken to crawl them also increases. Figure 6 shows the improvement gained in performance when executed using the middleware. As the number of workers increase, the execution time also decreases. These results indicate that the distributed solution scales linearly.

For measuring the execution time of MD5 hashing, the number of messages to be hashed by each thread is kept at a constant of five hundred messages. The performance is compared by executing the application on a single machine and using the middleware. Two and three workers are used for the purpose of comparison and time taken with five and ten threads is measured. Figure 7 shows the results obtained.

MD5 hashing is a CPU-intensive process. As can be seen in Figure 7, when the number of workers increase, the time taken to execute the application decreases. This is because the available CPU increases and hence a speed-up is obtained.

Comparison of Scheduling Heuristic for CPU-Intensive Application. In this section, the different scheduling algorithms

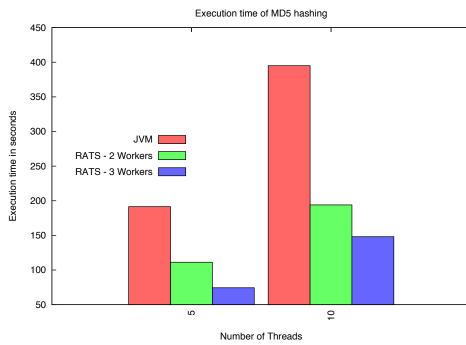

Figure 7: Execution time for MD5 hashing

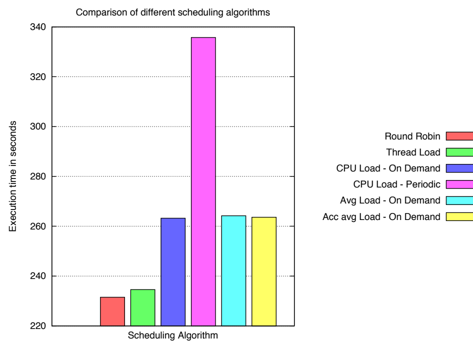

Figure 8: Execution time for high thread workload-Low dispersion of thread workload and low inter-arrival times

are evaluated with different application behavior. All the experiments are carried out with MD5 hashing of multiple messages and the application behavior is modified in order to classify based on its thread characteristics. To understand how different scheduling algorithms behave with different application characteristics, thread behavior is modified in an ordered fashion. The characteristics varied for the application are: load of each thread and thread inter-arrival times and is evaluated on a dedicated or uniform cluster where there are no other workloads.

The results obtained for high thread workload-Low dispersion of thread workload and low inter-arrival times are shown in figure 8 and low thread workload-Low dispersion of thread workload and

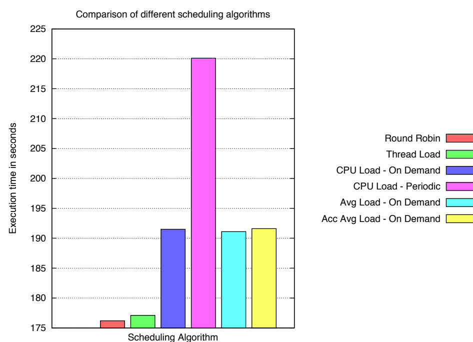

Figure 9: Execution time for low thread workload-Low dispersion of thread workload and low inter-arrival times

low inter-arrival times are shown in figure 9. Round robin performs better because the threads have equal workload and are equally spaced with their arrival times. Cpu load-on demand incurs the overhead of obtaining the load information from every worker before making a decision. The results obtained for low thread workload are similar, except that the the time taken for execution is considerably lower. This is because the workload is lesser. The combination of thread workload with inter-arrival times does not affect any of the scheduling algorithms and as a result the behavior of the scheduling algorithms remain the same.

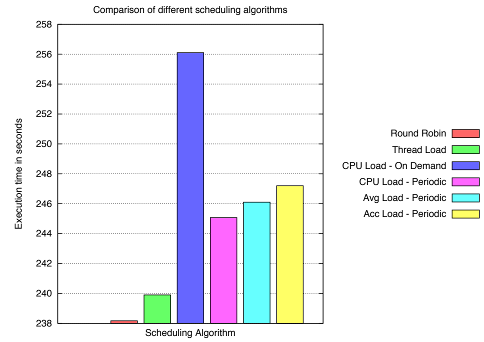

Figure 10: Execution time for high thread workload-Low dispersion of thread workload and high inter-arrival times

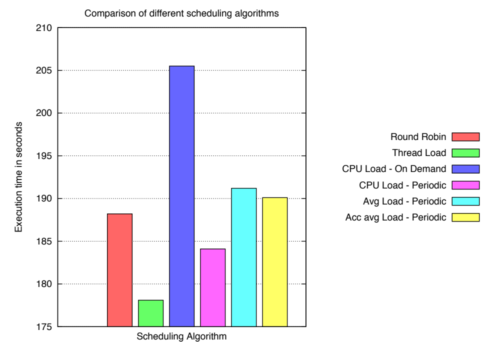

Figure 11: Execution time for low thread workload-Low dispersion of thread workload and high inter-arrival times

The application is then modified to make threads perform similar amount of computation with threads arriving after a large amount of time. In other words, the dispersion of thread work load is low and the time taken for arrival of threads is high. The results obtained for high and low thread workloads are shown in figure 10 and 11 . For high thread workload cpu load-periodic finishes faster than cpu load-on demand unlike the previous scenario. The lowest period is enough time to update the state information asynchronously as opposed to synchronous update for on-demand. For low thread workload, some of the threads finish its execution before all threads are scheduled and the thread load heuristic is thus able to make a better decision than round robin. The overhead incurred by monitoring the cpu load increases the time taken to schedule threads and hence cpu-load heuristic performs worse than round-robin and thread load heuristic.

Finally the application is modified to make the threads perform different amount of workloads. The workload is thus highly dispersed. The threads are made to arrive almost instantaneously accounting for low inter-arrival time and highly spread out accounting

for high inter-arrival time.The results obtained are shown in figure 12 and 13.

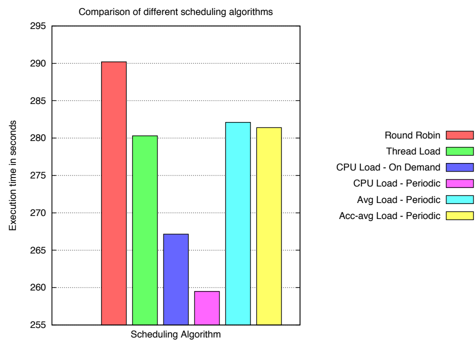

Figure 12: High dispersion of thread workload and high inter-arrival times

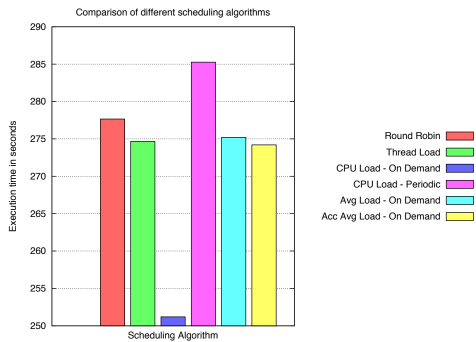

Figure 13: High dispersion of thread workload and low inter-arrival times

For high inter-arrival times, the scheduling heuristic cpu load-periodic consumes the least amount of time to finish execution. Although the scheduling heuristic has no information about the time taken to finish a job, some jobs finish much earlier than all the threads are scheduled. This information helps the heuristic make a better decision and spreads out threads of high and low workloads equally among the different workers. For low inter-arrival times, Cpu load -on demand performs better than any of the other scheduling heuristics, because the work load of threads are unknown and this heuristic aims to greedily equalize the cpu load of different workers as and when threads arrive. Cpu load-periodic takes a much higher time as the lowest possible period to update the state information is higher than the inter-arrival time between most of the threads. Most of the threads are hence scheduled on the same worker.

Non-uniform cluster. A non-uniform cluster is shared between multiple processes or users and can have other workloads running along with our middleware. This results in a varied load among different machines in the cluster. Figure 14 show a situation where Cpu-load metric may not always provide a correct view of the system. MD5 hashing application was executed on a cluster with two workers. Worker2 was already executing an I/O intensive application and worker3 was not loaded. Worker2 starts at a previous load of 0.3 while worker3 starts at a load of 0. The I/O intensive application remains idle during most of its cpu time as it either waits

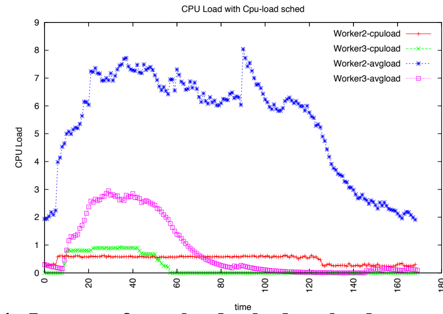

Figure 14: Impact of cpu-load sched on load average and cpu utilization

or performs an I/O operation. Because all the processes are scheduled by the processor for a definite quanta, the overall cpu usage is influenced by the I/O intensive process. Once the cpu-load of worker3 increases beyond that of worker2, threads are scheduled on worker2 till the loads of both these workers become equal but since the load of worker2 does not rise beyond 0.6, all consequent threads are launched on worker2. For applications with many number of threads, the scheduling can prove rather detrimental than useful as it considerably affects the response time. Load-Average and accelerated-Load-Average scheduling heuristic overcomes this problem and this is shown in figure 15 and 16.

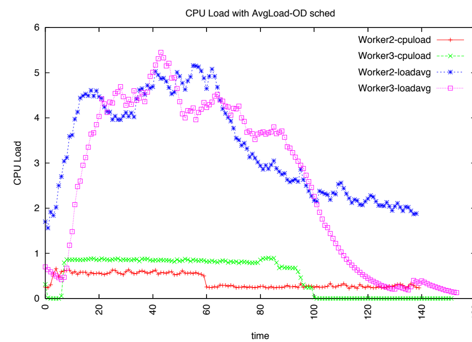

Figure 15: Impact of load-avg sched on load average and cpu utilization

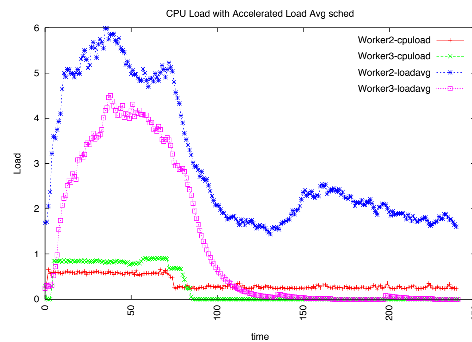

Figure 16: Impact of accelerated load-avg sched on load average and cpu utilization

Scheduling heuristic accelerated-load-average takes into account the instantaneous change in load averages caused by the application and thus performs better than load-average scheduling. Load-

average scheduling is very conservative as it always takes into account the previous load of the system. From the figure 15 it can be seen that worker2 finishes execution at around 60 while worker3 finishes execution only at time 100. This is because of the conservative nature of the scheduling heuristic and once the cpu load saturates, threads are launched conservatively on worker2 as it already has a high load average. This problem is mitigated by accelerated-Load-Average. From figure 16, it can be seen that the execution finishes earlier because more threads are launched on worker2 at the expense of tolerating minor differences in overall load average. Due to space constraints, tabulated classification of scheduling heuristics for different application characteristics is not shown.

Memory usage. In order to stress test the system for scalability in terms of memory, we developed a memory intensive application aimed to generate a synthetic memory load by creating as many objects as possible and simply iterate through them. In order to test the application, we allocated 7GB of memory for the JVM and ran the application on a single JVM and on top of our middleware using 2 workers. Each thread created an object with integer array of size 10000 and the size of integer is assumed to be 4 bytes. The results obtained are shown in Figure 17.

Figure 17: Memory stress test

It can be seen from the results that for a single JVM, approximately 4GB of memory was allocated and beyond that the system gave an out of memory error. But using the middleware, the system scales with respect to memory as the cluster now behaves like a virtual memory with the Terracotta server responsible for handling large heaps. As a result, we were able to allocate up to 15GB of data using the middleware. This result shows that the application scales in terms of memory.

Application Modeling. Based on the results obtained, we model application based on the characteristics of the cluster and the thread characteristics of the application. The cluster characteristics are classified into three categories: dedicated or uniform cluster, highly unbalanced cluster, and I/O intensive cluster. Table 3 classifies the application according to the most suited scheduling algorithm on a dedicated cluster and Table 4 classifies the application based on the previous load in a non-uniform cluster.

5. CONCLUSION

RATS middleware bridges the gap between transparency and efficient scheduling of threads using Terracotta to keep data consistent across the cluster and scale existing applications with ease. It supports multiple scheduling heuristics, each best suited for a specific thread behavior in an application. The thread behavior can be obtained by the profiling feature supported by the middleware. Based on the results obtained, a cpu-intensive application can be modeled based on the characteristics of the cluster and thread characteristics. The cluster characteristics is classified into two categories dedicated or uniform cluster and unbalanced or non-uniform cluster. An application with varying thread characteristics can be modeled for a dedicated cluster as per the results obtained through figure 8 to 13. Non-uniform cluster has the same behavior for on-demand and periodic updates of resource usage as that of uniform cluster. It is important to note that the behavior of scheduling heuristic round-robin and thread-load are unpredictable in a non-uniform cluster with any kind of application. Similarly with an existing I/O or network intensive load, the scheduling heuristic cpu-load becomes irrelevant and accelerated-load-average performs the best.

6. REFERENCES

[1] J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill, J. Andersson Ý, S. Weber Ý, E. Cecchet Þ, C. Jensen Ý, V. Cahill Ý, and Trinity College. Kaffemik - a distributed jvm on a single address space architecture, 2001.
[2] Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing*, pages 4–11, 1999.
[3] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.
[4] K. Etminani and M. Naghibzadeh. A min-min max-min selective algorihtm for grid task scheduling. In *Internet, 2007. ICI 2007. 3rd IEEE/IFIP International Conference in Central Asia on*, pages 1 –7, sept. 2007.
[5] Thomas Fahringer. Javasymphony: A system for development of locality-oriented distributed and parallel java applications. In *In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000*. IEEE Computer Society, 2000.
[6] H. Izakian, A. Abraham, and V. Snasel. Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, volume 1, pages 8 –12, april 2009.
[7] Yun-Han Lee, Seiven Leu, and Ruay-Shiung Chang. Improving job scheduling algorithms in a grid environment. *Future Generation Computer Systems*, 27(8):991–998, October 2011.
[8] Joao Lemos. Distributed clustering and scheduling of vms, master thesis.
[9] M. Maheswaran, S. Ali, H.J. Siegal, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 30 –44, 1999.
[10] Rajendra Sahu. Many-Objective Comparison of Twelve Grid Scheduling Heuristics. *International Journal*, 13(6):9–17, 2011.
[11] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
[12] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. pages 178–204. Springer-Verlag, 2002.
[13] R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed shared memory management for java. In *In Proc. sixth annual conference of the Advanced School for Computing and Imaging (ASCI 2000*, pages 256–264, 1999.

[14] Matthias Zenger. Javaparty - transparent remote objects in java, 1997.

Application Type	**Thread Workload**	**Inter Arrival time**	**Execution time of Scheduling Heuristic**
CPU Intensive	Low Dispersion, high load	low	RoundRobin ≈ Thread-load < Cpuload-OD < Cpuload-Periodic Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, low load	low	RoundRobin ≈ Thread-load < Cpuload-OD< Cpuload-Periodic. Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, high load	high	RoundRobin ≈ Thread-load < Cpuload-Periodic < Cpuload-OD. Load-avg and Acc-Load-avg (uncomparable)
	Low Dispersion, low load	high	Thread-load < RoundRobin < Cpuload-Periodic < Cpuload-OD. Load-avg and Acc-Load-avg (uncomparable)
	High Dispersion	high	Cpuload-Periodic < Cpuload-OD < Thread-load ≈ Load-avg and acc-Load-avg < RoundRobin
	High Dispersion	low	Cpuload-OD < Thread-load ≈ Load-avg and acc-Load-avg < Cpuload-Periodic < RoundRobin

Table 3: Application Modeling on a dedicated cluster

Previous Load	**Application Type**	**Execution time of Scheduling Heuristic**
Non I/O or network intensive	CPU intensive	CPU-load < Accelerated-Avg-Load < Avg-Load. (Others are irrelevant)
I/O or network intensive	CPU intensive	Accelerated-Avg-Load < Avg-Load. (Others are irrelevant)

Table 4: Application Modeling on a Non-uniform cluster