Enhanced Parallel Thread Scheduling for Java Based Applications on Multi-Core Architecture

Kam-Mun Chong
Faculty of Information Technology
Multimedia University
Cyberjaya, Malaysia
kmchong@mmu.edu.my

Hai-Shuan Lam
Faculty of Engineering
Multimedia University
Cyberjaya, Malaysia
hslam@mmu.edu.my

Chikkannan Eswaran
Faculty of Information Technology
Multimedia University
Cyberjaya, Malaysia
eswaran@mmu.edu.my

Somnuk Phon-Amnuaisuk
Faculty of Information Technology
Multimedia University
Cyberjaya, Malaysia
somnuk.amnuaisuk@mmu.edu.my

Abstract —**With the ability to execute more than one thread parallelly, the multi-core processor is claimed to deliver greater performance for multithreaded applications. The performance of thread scheduler in Java Virtual Machine has a profound impact on the performance of multithreaded Java applications. However, the lack of efficient thread scheduler in Java Virtual Machine creates burden in scheduling threads into parallel processors. Various enhancements on thread scheduler in JVM are proposed to boost up the performance of Java applications on dual-core processor. Affinity scheduling mechanism yields a 4% of throughput improvement. Priority scheduling mechanism helps in reducing the overhead incurred by 23.1%. These results shown the mechanisms suggested are efficient in delivering better performance on dual-core processor.**

Keywords — **Java Virtual Machine, affinity scheduling, priority scheduling, dual-core processor**

1. Introduction

Dual-core processor supports two concurrent threads execution at hardware level. This has increased the efficiency of multithreaded applications. With the support of multithreading at language level and portability, Java technology is very popular for web services development. Therefore, multithreaded Java applications are expected to perform better on dual-core technology. However, the byte-code interpreter Java Virtual Machine (JVM) serves as an important role in determining the performance of Java applications. Based on prior studies, optimization is necessary for boosting the performance of Java application on parallel processors [1].

Current thread scheduling mechanisms in some JVMs are not well optimized for handling threads on parallel processors [2]. Therefore, optimization on threading mechanism is essential in allocating the computing resources fairly and efficiently to the application threads. Moreover, by simply having more resources does not ensure higher performance. Prolonged idle time on the processor can lead to performance bottlenecks due to under-utilization of the computing resources. Hence, it is vital to reduce the idle time by managing parallel resources efficiently. Frequent thread migration among the Logical Processors (LP) is another factor for the performance degradation [3]. Overhead incurred during the thread migration can be reduced by assigning the thread to a particular LP according to certain criteria.

Time slicing is a common technique used to ensure no monopolization of CPU by any thread. Nevertheless, the drawback is no prediction can be made of how long the thread may take to run. Performance of thread with short jobs may get penalized with long waiting time. Compiler-supported quasi-preemptive scheduling is practiced in Jikes RVM [2]. Therefore, priority-scheduling mechanism is an easy approach in minimizing waiting time of threads with short jobs. This paper focuses on optimizing thread-scheduling mechanism in Jikes RVM on dual-core processor. The Operating System (OS) used is Fedora Core 4 [4]. From Figure 1, for each VP created by Jikes RVM, there will be a pthread associated. SPECjbb2000 is used for benchmarking JVM [5]. The objectives of applying approaches mentioned are to maximize the CPU utilization and throughput as well as to minimize waiting time respectively. The results of this project suggested various threading mechanisms to overcome the bottleneck of Java application running on dual-core processors.

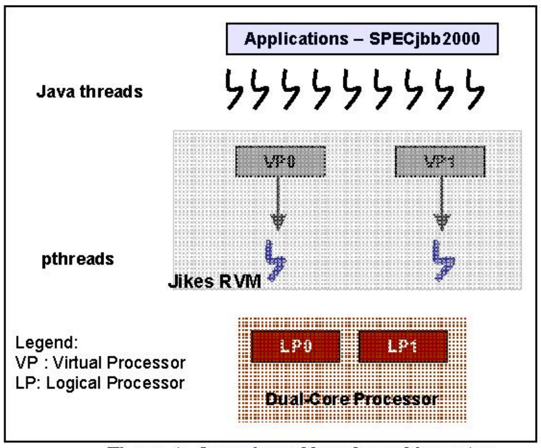

Figure 1: Overview of benchmarking setup

Many similar research works have been done on improving the thread scheduling mechanism in Chip Multi-Processor (CMP), Simultaneous Multithreaded (SMT) processor and Symmetric Multiprocessor (SMP) [6][7]. Similar studies were done in optimizing JVM, in terms of GC and thread scheduling. The best performing GC for SPECjbb2000 during high load and low load is Generational Mark Sweep collector (GenMS) and Copy Mark Sweep collector (GenCopy) respectively [8]. In thread scheduling, results show that in web services, finishing threads with lighter workload first as compared to the threads having higher workload achieved an improvement in performance [9].

2. Experimental Setup

Table 1: Hardware specification for dual-core PC

Hardware	Product Specifications
Processor Type	Intel Pentium D 930
Processor Speed	3.0GHz
BIOS	Intel D945GNTL Chipset
System Bus Speed	800MHz
System Memory Speed	667MHz
L2 Cache	2MB
Total Memory	1GB

Previous studies have shown that each type of workloads can be executed at certain optimum heap size. Hence, the heap size is set to 400MB for running SPECjbb2000 in this project [10]. As for the hardware setup mentioned in Table 1, eight warehouses found to yield an optimum results. Each warehouse is represented by a Java thread. The average result for default production configuration is collected. This set of results is needed for the performance comparison with the optimized thread scheduling algorithms. Three types of thread scheduling algorithms applied are namely affinity scheduling at Virtual Processor (VP) level, thread scheduling by binding VP to LP, and priority scheduling mechanism.

Affinity scheduling

Affinity scheduling is the process of binding thread to a particular processor. The purpose of introducing affinity scheduling is to minimize idle time of the processor. When the dependent threads are scheduled to two different processors, overhead will incur in retrieving the shared data between these two threads. This overhead is difficult to predict, therefore the results will only be valid if independent threads are considered. Application threads generated by SPECjbb2000 are by nature independent from each other. Therefore, affinity scheduling can be applied. Affinity scheduling is applied at VP level and LP level, namely VP affinity scheduling and CPU affinity scheduling respectively as shown in Figure 2.

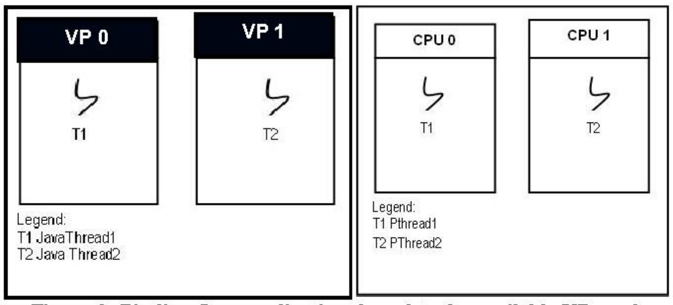

Figure 2: Binding Java application thread to the available VPs and binding pthread to available CPUs

VP Affinity Scheduling

In Jikes RVM production configuration, 8 types of daemon threads are available. Idle threads and Garbage Collection (GC) threads, by default are bound to the specific VPs. The rest of the six types are not affined to any VP. Therefore, in this context idle and GC threads are excluded from the daemon threads type. A profiling test was done on finding percentage ratio of time spent on daemon threads versus time spent on application threads. The results shown the percentage ratio is less than 1%, which is insignificant. Therefore, VP affinity scheduling will be applied to Java application threads only.

Experiment 1 focused on profiling the time spent on different types of VP scheduling. In this case, eight application threads from SPECjbb2000 are applied in this case. In Jikes RVM, *scheduleThread( )* method schedules thread into VPs. The algorithm of *scheduleThread( )* is shown in Figure 3. Time spent on various types of scheduling varies from one another. The outcome of this experiment helped to identify the performance bottleneck during VP scheduling.

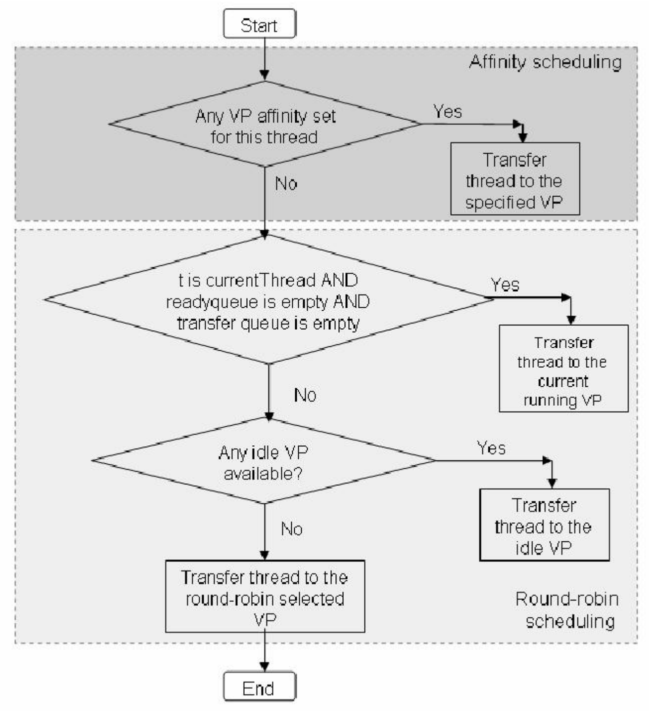
Start
Affinity scheduling
Any VP affinity set for this thread
Yes
Transfer thread to the specified VP
No
t is currentThread AND readyqueue is empty AND transfer queue is empty
Yes
Transfer thread to the current running VP
No
Any idle VP available?
Yes
Transfer thread to the idle VP
No
Transfer thread to the round-robin selected VP
Round-robin scheduling
End

Figure 3: Flowchart of scheduleThread( ) in default Jikes RVM

When one of the VPs finished the last runnable thread in the thread queue, it will be declared as idle VP and the idle thread will occupy it. This idle thread sends a request for work and then keeps waiting for a short time. If there is any extra runnable thread in another VP, this runnable thread will be transferred to the idle VP. The idle thread yields back to idle queue when work reaches and idle VP will then starts execution of the transferred thread. Yet, processor cycles are occupied by idle thread while waiting for the arrival of job. Frequent idle thread execution can prolong the total elapsed time of the entire application. This can penalize the throughput of the application. Reducing the number of idle thread executions will hence enhance the overall performance. Binding the application threads to VP can help minimizing the execution of idle thread. Figure 4 illustrates how the Java application threads are distributed evenly to the available VPs. The id of each VP will be assigned to the Java application thread interchangeably. The assignment is done at the JVM level, before these threads are scheduled into the thread queue. The assigned VP's id for the thread will remain the same during runtime.

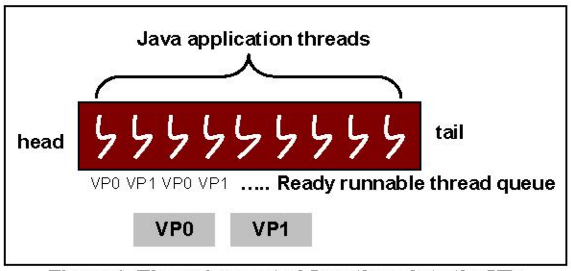

Figure 4: The assignment of Java threads to the VPs

CPU Affinity Scheduling

In Jikes RVM, number of LPs detected by OS, determines the number of VPs to be generated. Each VP generated is associated to a pthread. In this project, two logical processors are available. Hence, there are two pthreads. Since both the pthreads are allowed to execute in any LPs, there are possibilities for these two pthreads to be scheduled on the same LP. This leads to idle time on processor. This issue can be solved by applying CPU affinity scheduling, which ensures the two pthreads are attached to separate LP.

Priority Scheduling

The main aim of conducting Experiment 2 is to verify the suggested hypothesis. The hypothesis suggest that reducing the waiting time of the application thread during execution time can achieve better performance gain. In this experiment, a program with eight Java threads was tested. Each thread had to do iterative computations. During the first run, all the threads were started simultaneously. Elapsed time for each thread was recorded. For the second run, in order to avoid under utilization of the two LPs reside in the dual-core processor, two threads are started at the same time instead of one. All the threads are started together. The next two threads can only start execution if and only if the previous two threads are finished. Elapsed time for each thread was collected. For sequential running threads, the elapsed time of all threads by finishing the heavy-loaded threads first and vice versa is gathered for comparison as well. Priority scheduling is defined as the ordering of thread execution based on the predefined priority number. The solution of finishing the thread with light workload in a serial manner first is proposed to lessen the overhead incurred by the waiting time. To address this, priority scheduling is introduced. The thread with lighter workload is assigned a higher priority than the thread with heavier workload. The thread with higher priority must get finished first as compared to the thread with lower priority. The proposed idea is similar to the one proposed by Tai [9]. Mechanism suggested by Tai, is further enhanced in terms of thread sorting mechanism based on priority. The main objective is to eliminate the waiting time of each application thread.

Priority Scheduling Mechanism

Pseudojbb is used to benchmark the performance of Jikes RVM after applying priority mechanism. Pseudojbb is a variant of SPECjbb2000, where a fixed number of workload in multiple warehouses is executed. The priority mechanism mainly consists of three main parts. First part is the identification of short job threads and long job threads. Second part is the priority assignment to the threads. The third part is thread selection where threads with higher priority are selected for execution first. For the first part, the workload for each warehouse is fixed at application level as shown in Table 2. Therefore, Warehouse 1 and Warehouse 2 are defined as the heaviest threads, whereas Warehouse 7 and Warehouse 8 are defined as the lightest threads. Each thread is assigned a unique name to be passed to JVM.

Table 2: The assignment of workload to each warehouse

Warehouse	No. of Transactions
Warehouse 8 & Warehouse 7	9500
Warehouse 5 & Warehouse 6	95000
Warehouse 3 & Warehouse 4	900000
Warehouse 1 & Warehouse 2	950000

At JVM level, the mechanism is able to trace each thread according to the name of the thread. In this way, threads with heavy and light load can be identified clearly. Priority number in the range of 1 to 10 is assigned in the order as shown in Table 3 for all threads.

Table 3: The assignment of priority for each warehouse

Warehouse	Priority No.
Warehouse 8 & Warehouse 7	10 (Highest priority)
Warehouse 5 & Warehouse 6	7
Warehouse 3 & Warehouse 4	4
Warehouse 1 & Warehouse 2	2 (Lowest priority)
Daemon threads	5

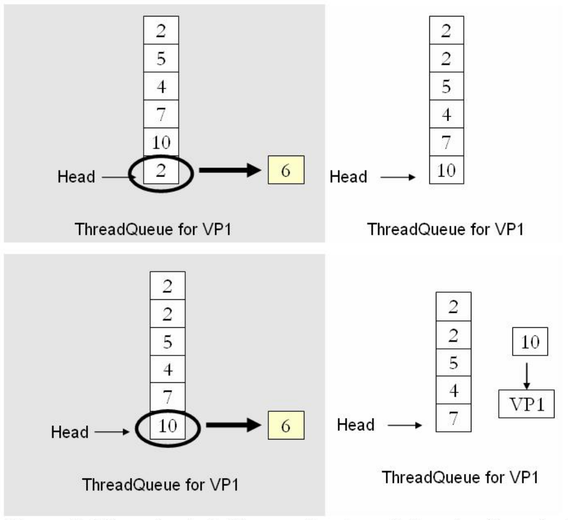

Figure 5: Thread scheduling mechanism of choosing thread with highest priority for execution

When VP is removing runnable thread from ready thread queue for execution, the thread priority number is checked if it is less than the predefined number. If the answer is no, this thread will be fetched by VP for execution. However, if the answer is yes, the thread will be queued back to the end of the ready thread queue. Next thread in the queue is selected for priority number comparing again. This process repeats. This ensures that thread with higher priority finished execution first. The thread selecting mechanism is shown clearly in Figure 5.

3. Results and Discussion

VP Affinity Scheduling

Experiment 1

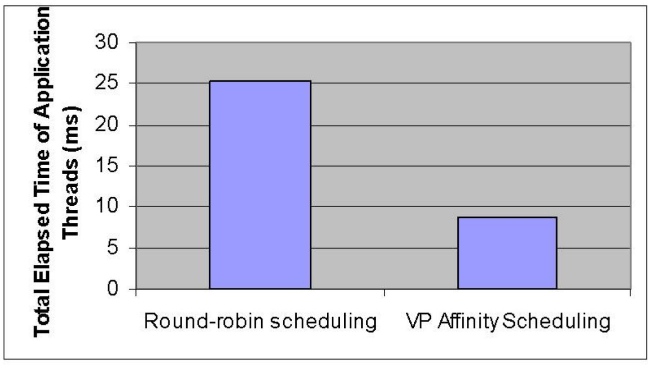

Figure 6: Graph for comparing total elapsed time round-robin scheduling vs. VP affinity scheduling for 8 warehouses

The time spent by scheduling thread to idle VP, round-robin selected VP, and to the same VP is added up. The sum mentioned is compared to the total time spent by scheduling thread to affined processor. The outcome in Figure 6 shown that execution time of eight application threads from SPECjbb2000 can be reduced by 65.3% after applying affinity scheduling at VP level.

Jikes RVM with the VP affinity scheduling outperforms the default round-robin scheduling by 4%. This is shown in Figure 7. Throughput for one warehouse and two warehouses are improved by 2.23% and 9.95% respectively. Starting from 3 warehouses to 8 warehouses the performance gain is in the range of 1% to 2%. A few reasons for the gain in performance for all warehouses, first is due to the reduction of idle processor declaration. Second reason is time consumed by scheduling to VP using affinity is much lesser than the sum of the time spent by default round robin scheduling to VP. The marked improvement for 2 warehouses execution can be explained by full exploitation of the resources resides in the dual-core processor. Applying affinity scheduling when thread number is greater than two, much time is wasted in waiting for the affined VP to be released for scheduling. When the thread number goes beyond the number of processor, a smart load balancing mechanism is needed to maximize the utilization of resources rather than affinity scheduling.

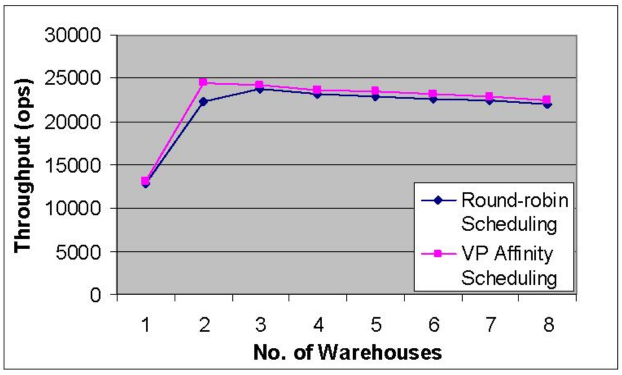

Figure 7: Result of VP affinity scheduling vs. round-robin scheduling up to 8 warehouses from SPECjbb2000

(CPU+VP) Affinity Scheduling

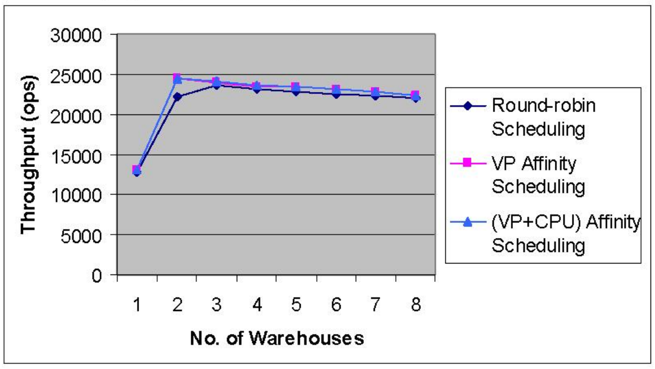

Figure 8: Result of comparing round-robin, VP affinity scheduling and VP combined with CPU affinity scheduling

Figure 8 reveals that after applying the CPU affinity scheduling on top of VP affinity scheduling, the overall performance is improved by 0.5% as compared to VP affinity scheduling only. Ranging from 1 warehouse to 8 warehouses, it brings less than 1% of enhancement as compared to application of VP affinity scheduling. This can be explained by without binding the VP bounded pthreads to LPs, the two VP bounded pthreads are scheduled evenly to two LPs. The two VP-bound pthreads can hardly get scheduled to the same LP. As in the system the CPU cycles occupied by daemon threads is insignificant as compared to both VPs.

A second benchmarking is conducted to show CPU affinity scheduling helps during concurrent execution of heavily loaded program and SPECjbb2000. The program consists of a thread performing iterative computations. The graph in Figure 9 shows clearly that without any CPU affinity scheduling the performance of SPECjbb2000 with eight warehouses is not stable. CPU affinity thread scheduling outperforms the default round-robin scheduling by 200.8%. The great difference between these two scheduling mechanisms can be accounted to the reduction of the waiting time spent for preempting heavily loaded thread in the program. When there are other programs running simultaneously with the multithreaded application, CPU affinity scheduling introduction ensures better performance of the application.

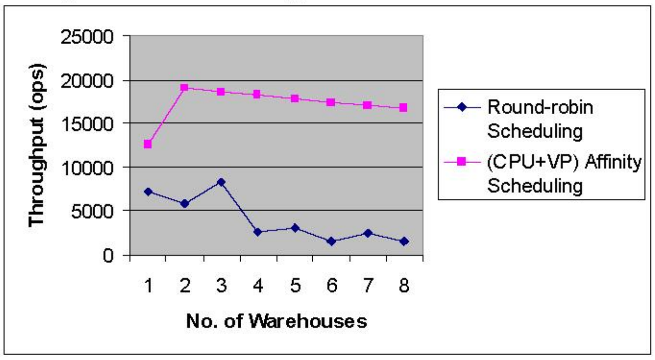

Figure 9: Results of (CPU+VP) affinity scheduling vs. round-robin scheduling by introducing program with heavy computation

Priority Scheduling

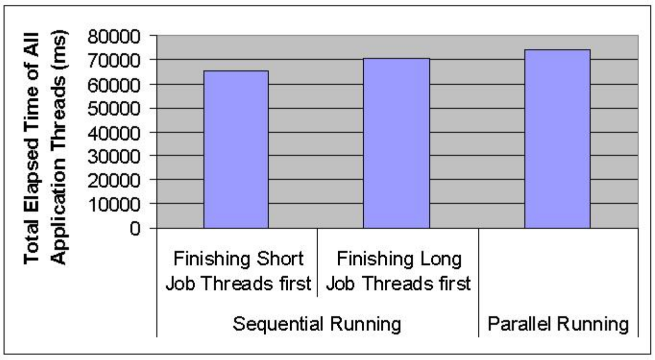

Figure 10: Comparing results of sequential running vs. parallel running threads

The time spent for sequential running threads is much shorter than parallel running threads as shown in Figure 10. The percentage of overhead reduction for sequential run that finishes short job threads first (A) as compared to parallel running is 13.2%. By comparing sequential run finishes long job threads first (B) against parallel running, the percentage of overhead reduced is only 5.2%. For (A), the time spent is shortened by 7.6% as compared to (B). The reduction in time consumed is mainly due to further minimization of the short job threads' waiting time.

The percentage of waiting time reduced for short job threads in (A) against short job threads in (B) is in the range of 5.6% to 25%. For short job threads in (A), the overhead is reduced by 9.82% to 272% as compared to short job threads in parallel running threads. This verifies the suggested hypothesis is correct. Finishing thread's execution one after another helps in reducing the waiting time spent in quasi-preemptive round-robin scheduling significantly.

Priority Mechanism

The overall percentage of overhead after applying priority mechanism that finishes short job threads first (C) is reduced by 23.1% as compared to parallel running threads. This is shown in Figure 11. However, for priority mechanism that finishes long job threads first (D) versus parallel running threads, the total overhead is minimized by 7.76%. For (C), the time spent is shortened by 14.2% as compared to (D).

For short job threads in (C), the overhead is decreased by 50% to 170% as compared short job threads in parallel running threads. However for short job threads in (C), the overhead is reduced by 20% to 100% as compared to short threads in (D). Priority mechanism, which finishes short job threads first, helps in minimizing the penalization of the short job threads. This reduces the elapsed time of the application threads drastically.

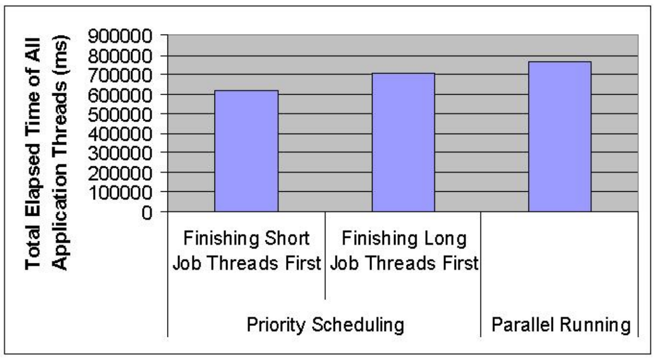

Figure 11: Performance of applying priority mechanism vs. default parallel running mechanism

4. Conclusion

This paper has demonstrated affinity thread scheduling at VP level can achieve an overall throughput improvement of 4%. With VP affinity scheduling mechanism, the Java application performs better when the application thread number is equal to the number of the processor. The reason for this is the utilization of the underlying CPU resources is maximized. When the number of threads is greater than number of processors, load-balancing mechanism is necessary. As this ensures less waiting time spent by threads to be scheduled to VP. During the introduction of one pthread with heavy job into system while running multithreaded Java application, CPU affinity scheduling achieved remarkable performance gain of 200.8%. The suggested priority scheduling mechanism which finishes short job threads first, helps in reducing the overhead of short job threads at least by 50%, thereby the elapsed time of all threads is shortened by 23.1% as compared to parallel running threads. These three approaches assist in maximizing the utilization of computing resources and throughput as well as reducing waiting time respectively.

REFERENCES

[1] Yau Meileng, H.S.Lam, G.S.V Radha Krishna Rao, C.Eswaran, "Effect of Hyper-Threading Technology on Java Virtual Machines: A Performance Study", The 9th IASTED International Conference on Internet And Multimedia Systems And Applications, EuroIMSA 2005, Grindelwald, Switzerland, pp.107-112

[2] Jikes RVM Homepage, http://jikesrvm.sourceforge.net/

[3] Brian J. Welch, "Impact of Load Imbalance on Processors with Hyper-Threading Technology", http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/hyperthreading/20477.htm

[4] Fedora Project, sponsored by Red Hat, http://fedora.redhat.com/

[5] The Standard Performance Evaluation Corporation. SPECjbb2000, http://www.spec.org/jbb2000/

[6] Michela Becchi, Patrick Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures", IEEE Conference On Computing Frontiers 2006, Ischia Italy, pp. 29-39

[7] Yen-Kuang Chen, Lienhart, R., Debes, E., Holliman, M., Yeung, M., "The impact of SMT/SMP designs on multimedia software engineering - a workload analysis study", Fourth International Symposium on Multimedia Software Engineering, 2002,pp. 336- 343

[8] Lam, Hai-Shuan; Rao, G.S.V. R.K.; Eswaran, Chikkanan; Ng, Kok-Seong,"Performance Comparison of Various Garbage Collectors on JVM for Web Services," Communications and Information Technologies, 2006. ISCIT '06. International Symposium on , Thailand, pp.711-715

[9] Lam, Hai-Shuan; Rao, G.S.V.R.K.; Eswaran, Chikkanan; Tai, Ewe-Shin, "Optimization of JVM by Dynamic Thread Prioritization for Web Services, "Communications and Information Technologies, 2006. ISCIT '06. International Symposium on, Thailand, pp.716-720

[10] KM Chong, HS Lam, Radha Krishna Rao, C.Eswaran, Somnuk Phon-Amnuaisuk, "Performance Optimization Of Java Virtual Machine On Dual-Core Technology for Web Services - A Study Proceedings of The 9th International Conference on Advanced Communication Technology, 2007, Phoenix Park, Korea. pp 567- 570