Performance Enhancement and Prediction Model of Concurrent Thread Execution in JVM

Khondker Shajadul Hasan

1Department of Computing Sciences, University of Houston - Clear Lake, 2700 Bay Area Blvd., Houston TX 77058, USA, HasanK@UHCL.edu

Abstract - Performance of a Java Virtual Machine (JVM) is quantified in terms of the JVM’s relative CPU availability at executing concurrent Java threads. The total CPU loading of a JVM is defined by the sum of the CPU utilization factors of all threads executing on the JVM. Sharp performance degradation has been observed while JVM executes concurrent threads with exactly same CPU load. An analytical model has been proposed and implemented to improve the scenario. Extensive experimental studies and statistical analysis are performed to validate the performance enhancement of concurrent thread execution and provide a basis for an empirical model for improving CPU performance. To facilitate scientific and controlled empirical evaluation, synthetically generated threads are employed that are parameterized by their CPU utilization factor, which is defined as the fraction of time a thread spends utilizing CPU resources.

Keywords: Concurrent threads; CPU efficiency; Execution Efficiency, Java Virtual Machine.

1. Introduction

The primary contribution of this section is to introduce the thread execution behavior of this empirical model. Threads are considered as independent tasks, meaning there are no interdependencies among threads such as message passing. A thread is modeled by a series of alternating work and sleep phases. For the purposes of this study, the work portion of a phase is CPU-bound and requires a fixed amount of computational work (i.e., CPU cycles). The sleep portion of a phase does not consume CPU cycles, and its length relative to that of the work portion is used to define the CPU load usage factor for a thread. Figure 1 shows three work-sleep phases of a thread.

In this analytical model, a thread in the work portion of a phase will remain in the work portion until it has consumed enough CPU cycles to complete the allotted work. After completing the work portion of the phase, the thread then enters in the sleep portion where it sleeps (does not consume CPU cycles) for an amount of time defined by the CPU utilization factor. When multiple threads are spawned concurrently, the JVM runs those threads in a time sharing scheduling technique [2]. The performance (and availability) will be degraded when the work phases of all threads overlap each other in time. Figure 2 depicts a scenario where 3 threads with identical work and CPU loads are executed in a single-core execution environment. Here, each thread gets a maximum of 1/3 of the available CPU during their work phase, resulting in a work phase length 3 times wider than a single thread scenario.

Alternatively, if the work phases of the three threads are staggered to where there is no overlap, then there is no contention for the CPU resource and the CPU efficiency is essentially perfect. That is, all the work phases of concurrent threads are separated so that each thread can get the full attention of the CPU.

2. Motivation of Work

This section of the paper focuses more on the motivation behind the empirical studies to demonstrate the concurrent thread performance enhancement.
Besides motivation, this section also discusses the surroundings of the problem associated with executing concurrent threads with the same CPU load on JVM. The CPU performance of JVMs for executing synthetically generated concurrent threads is evaluated through experimental studies. The important objective of this empirical study is to determine the CPU performance of JVM handling concurrent threads. The system that is used for handling the single core test cases is an Intel Xeon CPU E5540 @ 2.53GHz clock speed, 1,333 MHz bus speed and 4 GB of RAM. All benchmark and experimental programs are implemented in Java language and JDK 1.6 has been used for execution.

Threads are given $2.0 \times 10^8$ units of work load to accomplish in 50 quantums. A quantum consists of a work phase and a sleep phase. Each thread needs to accomplish $4.0 \times 10^6$ units of work in each work phase so that it can complete total work in 50 phases. Sleep phase lengths of the threads are calculated depending on CPU load and remains constant during its life time, but the work time can vary depending on CPU availability. When threads have completed their work, the report of thread execution containing start time, work time, sleep time, number of phases, and end time are stored into multiple files for statistical analysis.

When two concurrent threads are spawned in a single core machine with the same amount of work, the thread execution time depends on the CPU load. It has been experimentally found that if those two threads have exact same CPU load, the performance is dreadfully poor. As the work phases of both threads lined up on top of each other, each thread gets a maximum $1/2$ of the available CPU which eventually doubles the work time. Moreover, both threads are working at the same time and sleeping at the same time which wastes the CPU resources as well. To validate this statement, several empirical studies have been conducted by spawning two threads concurrently in a single core machine with the exact same CPU load. Figure 3 shows execution times of thread 1 and 2 for multiple test runs.

Benchmark programs have been used to measure the ideal processing time for synthetic threads with different CPU loads. The benchmark processing time of one thread in a single core machine with a CPU load of 0.1, which is equal to 10% CPU usage, and work load of $2.0 \times 10^8$ units is 58,661 ms. For next test case, when two concurrent threads are spawned in JVM containing same CPU and work loads of 0.1 and $2.0 \times 10^8$ units, the average processing time has sharply increased to 99,842 ms for thread 1 and 100,824 ms for thread 2. Figure 3 shows the performance of those two concurrent threads.

As the work phases of both threads are staggered, the available CPU is shared among the threads which results in a larger work time which is around 72.4% more compared with the benchmark work time. Figure 4 shows test results involving 2, 3, 4, and 5 concurrent threads containing the same CPU and work load.

As the CPU loads are the same, work phases of all concurrent threads were staggered on top of each other. The available CPU is shared among all threads which have resulted in deprived performance. Figure 4 shows the increase of execution time as the number of thread increases though the cumulative CPU load is much lower than 100% CPU load. For each scenario, several test runs have been conducted and the average work time has been taken to plot the graph. Though the aggregate CPU load is 50% for 5 threads, the work time has risen to 162,754 ms which is around 275% higher than the benchmark work time.
A different empirical study has been conducted to verify the above findings involving two concurrent threads. In these tests, the CPU utilization of thread 1 has been varied from 0.05 to 0.5 (5% to 50%) but thread 2 remains a constant 0.1 unit. In Figure 5, the horizontal axis represents CPU load and the vertical axis represents execution time in milliseconds. The test results show that the execution time of thread 2 is expectedly always closer to the benchmark work time except in the case when thread 1 and thread 2 has exactly the same CPU load of 0.1. That is, when the CPU load of thread 1 reaches the same 0.1 like thread 2, the work time of thread 2 increases sharply from benchmark work time of 58,661 ms to 100,824 ms. For all other cases, work times of thread 2 are acceptably closer to the benchmark work time confirming that when work phases of both threads are staggered, the performance of both threads degrades significantly.

Next, to verify the dependability of this finding, another similar test case has been conducted with several runs by assigning different CPU load values for the first thread. In this test case, CPU load of the first thread has been varied from 0.05 to 0.5 units but CPU load of the second thread remains a constant like previous but with a value of 0.2 unit. Figure 6 also shows a similar finding where the work time of the second thread is very close to benchmark work time except the case where CPU load of both threads are 0.2. Benchmark work time for CPU load of 0.2 is 29,642 ms but when thread 1 and thread 2 have 0.2 CPU load, the processing time for thread 1 increases to 50,154 ms and thread 2 reaches to 49,757 ms.

In Figure 5, it can be seen that when the CPU load of thread 1 was 0.2, the processing time was 35,569 ms but in Figure 6, it has increased to 50,154 ms which is approximately 70.91% increase in execution time. Similar case studies have been conducted considering all possibilities by employing different CPU load for thread 2. It has been found for all the cases that when both threads have the same CPU load, the work time increases noticeably which degrades the thread execution efficiency.

### 3. Empirical Studies for the Model

In this section, an analytical model has been developed for enhancing CPU availability associated with executing concurrent threads containing the same CPU load on JVM. Several empirical studies have been conducted to determine the breach among the concurrent threads’ CPU loads to avoid possible work phase overlap. It has been found from test results that instead of having the same CPU load in all threads, having a slight variation (0.005 ≤ ε ≤ 0.1) can noticeably increase the performance of thread execution. Reduced gap value has been assigned iteratively to find out the minimum value which can separate the work phases. Several test results suggests that even when the thread utilization percentage is separated by 0.005 (0.5%), appreciable performance can be achieved. Multiple test cases have been conducted by applying this value to avoid work phase overlap.

In this test case, CPU load of thread 1 is varied from 0.09 to 0.11 but the CPU load of thread 2 is always 0.1. That is, CPU load value of thread 1 is varied 0.005 in each cases but thread 2 always remains 0.1 unit. Data from several test runs have been used to plot the graph of Figure 7 showing the execution time for both threads. In Figure 7, it can be seen that except for the case where thread 1 and thread 2 has exactly the same
CPU load of 0.1, it provides expected performance. That is, several test case results suggest that a variation of as small as 0.5% of CPU load can result in improved performance close to benchmark time.

Figure 7: Processing times of 2 concurrent threads for various CPU load.

This empirical study has been conducted to verify whether the above findings hold for two concurrent threads with the same work and CPU loads. Here thread 1 has been assigned 0.095 CPU load and thread 2 has been assigned 0.1 CPU load. That is, the CPU load gap between these two threads is 0.005 or 0.5%. The aggregate load in this case is 0.195 which is below the aggregate load of the previous case in which it was 0.2. Several test cases have been conducted for accuracy and the test result data have been used to plot the graph of Figure 8. In Figure 8, it can be seen that there are no sudden increases in work time for these threads. Moreover, their work time is close to benchmark work time. That is, with a slight variation in CPU load, the execution time is much less compared to the previous case even though the cumulative CPU load is less. These test cases results indicate that the approach of variation in work load has resulted in improved work time though the combined CPU load is less.

Figure 8: Processing time comparison of two concurrent threads with a slight CPU load variation of 0.005 units. Thread 1 has 0.095 and thread 2 has 0.1 CPU load.

Similar case studies have been conducted by spawning 3, 4, and 5 concurrent threads containing the same 0.1 CPU load in a single core machine which depicts a similar pattern compared with the 2 thread scenario in Figure 3. Figure 9 shows a four concurrent threads scenario with a CPU load of 0.1 each. The test results found that the average processing time of threads are 151,288 ms which is approximately three times higher than the benchmark processing time.

Figure 9: Processing time comparison of 4 concurrent threads with same CPU load of 0.1 units.

The next case study has been conducted for four concurrent threads to verify whether it is possible to achieve closer to benchmark execution time performance by separating 0.5% CPU load. So, in this test case, 4 concurrent threads are spawned but their CPU loads are separated by 0.005 unit. Concurrent threads are assigned 0.09, 0.095, 0.1, and 0.105 units of CPU loads respectively. Here, the total load is 0.39 which is less than the previous test cases’ aggregate load. Test results of several run shows that the average work time is 68,583 ms which is much lesser compared with the previous case where it was 151,288 ms. So, with a slight variation in CPU load, the processing time is very close to the benchmark program processing time. That is, with a small variation in CPU load among threads, the scheduling of thread execution helps to reduce thread work phase overlap.
Figure 10: Work time comparison of 4 concurrent threads with a slight CPU load variation of 0.005 units from 0.09 to 0.105.

As a large number of empirical studies have been conducted to validate the findings of performance enhancement, the test results of multiple scenarios of concurrent threads are presented in the following table. It shows an outstanding improvement in execution time when the CPU load is varied by 0.005 units which is exactly 0.5% of CPU load. When the CPU load for concurrent threads are varied as little as 0.5%, the execution time is very close to the benchmark execution time. Whereas, when all concurrent threads have the exact same CPU load, their work phases are staggered on top of each other which results in a very large execution time, 2 to 3 times more than the benchmark execution time depending on the number of concurrent threads. Figure 11 shows a complete scenario of concurrent threads processing time comparison among the benchmark, same CPU load, and slight variation of 0.5% in CPU load. This finding clearly suggests that the exactly same CPU loads for concurrent threads are not a good choice.

Table 1: Execution time for concurrent threads with exactly same CPU load and slightly different CPU load of 0.005 units.

<table>
<thead>
<tr>
<th>Thread Count</th>
<th>Same CPU Load Time (ms)</th>
<th>Slightly different CPU Load (0.5%) Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Thread</td>
<td>58,661</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>2 Threads</td>
<td>100,338</td>
<td>63,110</td>
</tr>
<tr>
<td>3 Threads</td>
<td>128,838</td>
<td>63,931</td>
</tr>
<tr>
<td>4 Threads</td>
<td>151,288</td>
<td>68,583</td>
</tr>
<tr>
<td>5 Threads</td>
<td>162,754</td>
<td>69,464</td>
</tr>
</tbody>
</table>

Figure 11: Performance of thread execution with a slight variation of CPU load and with exactly the same CPU load.

Another empirical study for 2 concurrent threads has been conducted in which both the work and sleep times are constant throughout their life cycle. The work and sleep phase lengths are calculated during runtime depending on CPU load. The amount of work that can be accomplished in each phase will vary depending on the CPU availability. When the work phases of threads are completely out of phase, threads can accomplish the highest amount of work in that stage. Figure 12 shows a case where work phases are completely out of phase.

Figure 12: Work phases of three concurrent threads are separated so that each thread can get the full attention of the CPU.

If the work phases of the three threads are staggered to where there is no overlap, then there is no contention for the CPU resource and the CPU efficiency is essentially perfect. That is, all the work phases of concurrent threads are separated so that each thread can get the full attention of the CPU. On the other hand, when the work phase of threads are perfectly staggered on top of each other, shown in Figure 2, the available
CPU will be shared among the active threads which will result in less amount of work accomplished in those phases.

Here, two threads are spawned concurrently with the same amount of CPU and work load. As these threads are CPU bound and starting with the work phase, the available CPU will be shared among the concurrent threads. Figure 13 show that when work phases of the threads are overlapped, the amount of work that has been accomplished is approximately 50K units. The work accomplishment varies based on the degree of work phase overlap. If threads are perfectly overlapped, then the work accomplishment can drop as low as 18K unit. When they are not overlapped, it can grow as large as 160K unit.

![Figure 13: Variable work accomplished by concurrent threads with the same CPU load.](image1)

To improve the scenario, a slight variation in the CPU load has been applied and several empirical studies have been conducted to analyze the outcome. Here, two concurrent threads have been spawned with the exact same amount of work but the CPU load has been varied slightly. Thread 1 has a CPU load of 0.1 whereas the CPU load of thread 2 has slightly been changed to 0.095 to conduct the test. The test results show that this slight variation (0.005 ≤ E ≤ 0.1) in CPU load has resulted in improved processing time. The start time difference between the work phases of threads has increased constantly which separated the work phases and has allowed threads to acquire full CPU attention to process its allocated job quickly. Figure 14 shows increase of time difference between threads’ start times.

![Figure 14: Start time difference between work phases of threads grows from cycle to cycle.](image2)

A further analysis of start time difference between the threads shows that there are few cases where the thread work phase has overlapped as well. Here, the quantum is 1000 ms. When the time difference between thread 1 and thread 2 work phase crosses 1000ms, the thread 2 work phase starts overlapping with the thread 1’s work phase of the next quantum. By calculating the start time difference, 1000-MOD (Start time difference, 1000), the work phase overlap can be seen. Figure 15 shows the work phase overlap among phases.

![Figure 15: Threads work phase start time difference in a scale of 100ms.](image3)

The positive outcome in this model is that the numbers of work phase overlaps are very small in number compared with the previous scenario. This small number of work phase overlap enables the threads to accomplish the highest amount of work. The work accomplishment in each work phase has increased to around 161 K unit for each thread except the overlapped phases. Figure 16 shows the work accomplishment of the concurrent threads. The combined work accomplishment by both threads in each work phase is around 317K units on average compared with the previous scenario where the average work accomplishment per phase was only 183K units.
This improved model has taken 63 phases to accomplish the total work instead of 110 phases taken by the previous setup. The context switching overhead has resulted in poor work accomplishment for the previous scenario.

Figure 16: Work accomplishment of threads with slight variation in CPU load.

Figure 16 also depicts that as the work phase overlaps in 1 to 2, 21 to 22, 40 to 41, 59 to 61 cycles, the work accomplishment drops to around 210K unit. For all other cases, work phases have not overlapped which enables them to get full attention of the CPU to achieve maximum amount of work.

4. Conclusion

This paper has presented an analytical model (and conducted empirical studies) for predicting (and measuring) CPU performance of JVM for handling synthetically generated concurrent threads. As observed, degradation in CPU performance occurs when concurrent threads has exactly same CPU load. It is more effective when the total CPU loading is less than the total capacity of all CPU cores. In addition to total CPU loading, the number of concurrent threads is a factor in predicting CPU efficiency; more threads generally incur more context switching overhead, which results in degraded efficiency. When the total load is less than the total capacity of all cores, the relative alignment of the working and sleeping phases of the threads can have a significant impact on CPU performance. Specifically, increased overlap of the work phases implies lower performance. It was demonstrated that shifting the relative phasing of the threads, using a slight variation in CPU load, to reduce possible work phase overlap can improve the performance (i.e., CPU efficiency). Random aggregate load values for concurrent threads were assigned for each for an extensive number of experimental measurements. A thread performance efficiency chart has been plotted to compare measured performances among the processing time of benchmark program, threads with equal CPU load, and threads with slightly different CPU load (0.005 ≤ ϵ ≤ 0.1). Two different approaches have been adapted and executed to prove the validation of this empirical model. These empirically measured efficiency values are the indication of applying slight variation for CPU bound threads. An empirical model for CPU performance presented in this paper can play a vital role for making critical CPU load decisions for most CPU bound commercial applications.

5. References


