2015

Factors Affecting Scalability of Multithreaded Java Applications on Manycore Systems

Junjie Qian
University of Nebraska-Lincoln, jqian@cse.unl.edu

Du Li
Carnegie Mellon University, duli@cs.cmu.edu

Witawas Srisaan
University of Nebraska-Lincoln, witty@cse.unl.edu

Hong Jiang
University of Nebraska-Lincoln, jiang@cse.unl.edu

Sharad C. Seth
University of Nebraska-Lincoln, seth@cse.unl.edu

Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on
DOI: 10.1109/ISPASS.2015.7095800

Factors Affecting Scalability of Multithreaded Java Applications on Manycore Systems

Junjie Qian*, Du Li†, Witawas Srisa-an*, Hong Jiang* and Sharad Seth*
*Department of Computer Science & Engineering, University of Nebraska-Lincoln, {jqian, witty, jiang, seth}@cse.unl.edu
†School of Computer Science, Carnegie Mellon University, duli@cs.cmu.edu

***Abstract*—Modern Java applications employ multithreading to improve performance by harnessing execution parallelism available in today's multicore processors. However, as the numbers of threads and processing cores are scaled up, many applications do not achieve the desired level of performance improvement. In this paper, we explore two factors, lock contention and garbage collection performance that can affect scalability of Java applications. Our initial result reveals two new observations. First, applications that are highly scalable may experience more instances of lock contention than those experienced by applications that are less scalable. Second, efficient multithreading can make garbage collection less effective, and therefore, negatively impacting garbage collection performance.**

I. Introduction

Developers of Java applications adopt multithreading to achieve higher performance by utilizing multiple processing cores. As such, it is important that these applications *scale well*; that is, the performance of an application should increase as more threads and more processing cores are employed. This performance improvement continues to occur until the time spent on sequential portion of the program outweighs the improvement achieved through parallelism. Because Java is a managed programming language, the performance of a Java application is determined by two performance factors: the time spent in application execution (*mutator* time) and the time spent in execution of runtime systems such as garbage collection (*GC* time). Furthermore, these two factors can also affect the scalability of a Java application.

In this paper, we conduct an investigation to reveal factors that can affect scalability of Java applications. Our study simultaneously considers both mutator and GC times to reveal insights on how each can contribute to the overall scalability of an application. First, we measured the application level lock contention with different numbers of threads. The results show that for scalable applications, lock contention increases with the number of threads but for poorly scalable applications, lock contention remains essentially constant as the number of threads increases. This implies that performance improvement achieved through parallelism in scalable applications outweighs the overhead due to higher instances of lock contention. Second, we characterized object lifespans and uncovered that higher execution parallelism can cause objects to live longer. Longer object lifespans degrade GC performance because most of the current GC techniques, including those based on the notion of generations, are most effective when objects die young.

II. Methodology

A. Metric We used object lifespan as a metric to help explain garbage collection effectiveness. We measured the lifespan of an object by observing the amount of heap memory that has been allocated to other objects between its creation and its death.

B. System setup We conducted our experiments on a NUMA machine with four AMD 6168 sockets each containing 12 processing cores with 64 GB RAM. We used OpenJDK 1.7 HotSpot as the Java Virtual Machine (JVM). The JVM was configured to use the stop-the-world throughput-oriented parallel garbage collector. We adopted Elephant track [1] as the object trace profiling tool; it produces an in-order trace of events pertaining to each object. In addition, we used Dtrace to profile lock usage, from which instances of contention during execution could be analyzed.

C. Benchmarks We choose 6 multithreaded applications from Dacapo-9.12 [2], *sunflow*, *lusearch*, *xalan*, *h2*, *eclipse* and *jython*. Each application instantiates about the same number of objects and requires same heap space even as we increase the number of threads. We then executed these applications under different thread and processor settings. The result suggests that we can characterize the first three applications as scalable and the remainder as non-scalable. In a scalable application, its execution time would reduce with more threads and more cores.

We then ran these applications by setting the heap size to three times the minimum heap requirements (i.e., if the heap is any smaller, the application would fail to execute). This is a common approach that has been used to evaluate GC performance. We varied the number of application threads based on the number of enabled processor cores (if we enabled four cores, we also set the number of threads to four). Because many helper threads also run concurrently with the application threads, the number of threads running on the real system is usually more than the available processor cores. However, most helper threads are short lived.

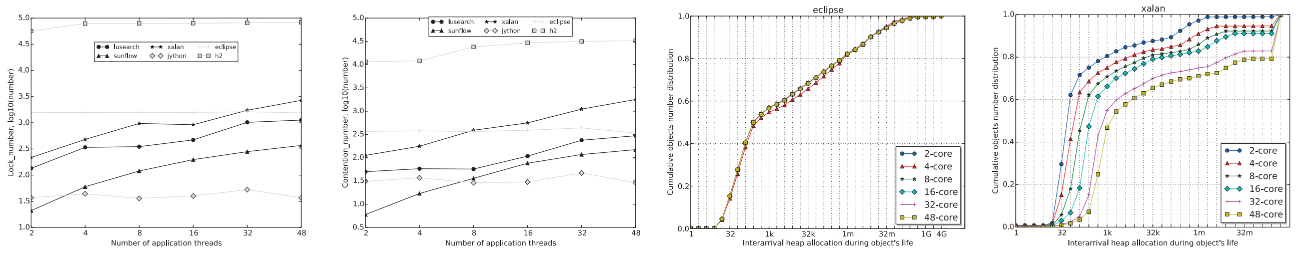
III. Results

In this section, we present the results of our investigation into the impacts of thread scaling on lock usage, lock contention, and object lifespan. We also report the observable impacts of GC on scalability.

In terms of workload distribution, the non-scalable applications employ only a small number of threads to perform the work. For example, *jython* mainly uses three to four threads to do most of the work even when we set the number mutator threads to be larger than 16. On the other hand, *xalan*, *lusearch*, and *sunflow* show nearly a uniform distribution of workload among threads. As we employed more threads, each thread would perform proportionally less work.

A. Lock Usage

Figure 1a and 1b illustrate lock acquisitions and instances of contention that occur with different numbers of threads. The different behaviors of the scalable and non-scalable applications are clearly evident: scalable applications show increasing lock usage and contention as the number of threads grows. On the other hand, lock usage and contention in non-scalable



(a) number of locks (b) number of lock contentions (c) eclipse's object lifetime (d) xalan's object lifetime

Fig. 1: Comparison of scalable and non-scalable applications on number of lock, number of contentions and object lifetimes.

applications remain unaffected by the number of threads. This is because scalable applications can better divide the workload among threads to achieve good performance gains. This is, despite of, having more instances of contention that can increase the synchronization overheads.

B. Object Lifespan

Because GC techniques, in general, are more effective when most objects die young, prolonged object lifetimes can have negative impacts on both GC effectiveness and GC performance. Figure 1c and 1d demonstrate the striking differences in the lifespan characteristics of objects in scalable and non-scalable applications. For example, *xalan*, a scalable application, has over 80% of objects with lifespans of less than 1KB when 4 mutator threads are used. When we increased the number of threads to 48, only 50% of objects have the lifespans of less than 1KB. On the other hand, *eclipse*, a non-scalable application, shows almost no change in object lifespans as we changed the numbers of threads from 4 to 48. The prolonged lifespans cause more objects to survive the nursery collection. Therefore, more time is spent copying these surviving objects to the mature generation and eventually resulting in more full GC invocations as the mature region is filled up more quickly. When a thread is suspended, it does not use the objects that it has created in the heap so they continue to stay alive. However, other threads continue to allocate objects in the heap, which is shared by all threads in an application. This would eventually result in a garbage collection invocation. In scalable applications, threads tend to share workload evenly; therefore, there is a greater competition for processors, resulting in longer wait time for a thread in the suspend state. This can prolong the lifetimes of objects created, but not yet used by that thread.

C. Impact of GC

Figure 2 reports the mutator times and GC times of the three scalable applications. As mentioned earlier, the heap usage and objects allocation pattern in these applications are not sensitive to the number of threads. Hence, one would expect very little impact on the GC performance as we increased the number of mutator threads. However, the figure shows that this is not the case. There are two important points to take away from this study. First, GC overhead keeps increasing as we increase the number of threads. Second, if we ignore the GC time, the mutator time would continue to be reduced as we scaled up the numbers of threads and cores all the way to 48 (as shown in Figure 2). This observation implies that GC can affect the overall scalability of these three applications.

IV. Conclusion and Future Work

In summary, execution parallelism can improve performance by having more threads jointly performing work.

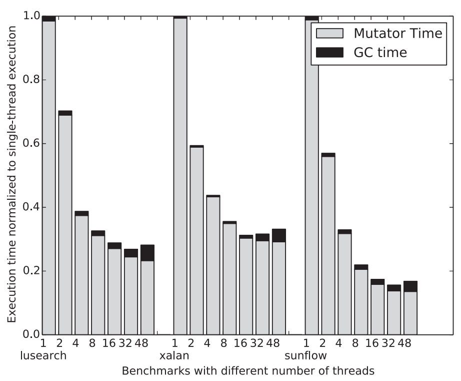


Fig. 2: Distribution of mutator and GC times

However, such collaboration also makes threads compete for resources such as processors and heap space. In scalable applications, workload can be divided evenly among threads, and therefore, it is beneficial to employ more threads. However, doing so requires careful synchronization of shared resources that can lead to more lock acquisitions and instances of contention. In addition, we also show that object lifetime is also affected as the heap usage is an aggregation of objects needed by both active and suspended threads.

Based on the results reported in this work, we make two suggestions to improve scalability of Java applications by focusing on JVM and OS implementation. First, we can bias scheduling to reduce lifetime interference; that is, worker threads are scheduled at the different phases of the execution to reduce competitions for heap and locks. Second, we can create compartmentalized heap to isolate objects from lifetime interference, which can potentially improves throughput performance in large multi-threaded server applications by reducing memory requirement and shortening garbage collection pause time.

Acknowledgment

This material is based on research sponsored by DARPA and ARO under agreement number FA8750-14-2-0053 and W911NF-13-1-0154, respectively. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

References

[1] N. Ricci, S. Guyer, and J. E. Moss, "Elephant Tracks: Portable Production of Complete and Precise GC Traces," in *ISMM*, 2013.

[2] S. Blackburn, R. Garner, C. Hoffmann, *et al.*, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *OOPSLA*, 2006.