A Study of Data Structures with a Deep Heap Shape [*]

Haggai Eran   Erez Petrank
Technion - Israel Institute of Technology
{haggaie,erez}@cs.technion.ac.il

Abstract

Computing environments become increasingly parallel, and it seems likely that we will see more cores on tomorrow's desktops and server platforms. In a highly parallel system, tracing garbage collectors may not scale well due to deep heap structures that hinder parallel tracing. Previous work has discovered vulnerabilities within standard Java benchmarks. In this work we examine these standard benchmarks and analyze them to expose the data structures that make current Java benchmarks create deep heap shapes. It turns out that the problem is manifested mostly with benchmarks that employ queues and linked-lists. We then propose a new construction of a lock-free queue data structure with extra references that enables better garbage collector parallelism at a low overhead.

Keywords Parallel garbage collection, Concurrent data structures, Linked-lists

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Memory management (garbage collection); E.1 [*DATA STRUCTURES*]: List, stacks and queues

General Terms Languages, Performance, Algorithms.

1. Introduction

In the past few years multi-core computers have become ubiquitous, and future computers are expected to be more and more parallel. Programmers are required to adjust in order to take advantage of modern and future hardware. But an interesting question is whether the systems and runtimes can scale to allow efficient executions on many cores. In this work we focus on the memory management aspect of the system and in particular, garbage collection (GC).

Modern programming languages such as Java^{TM} and C# use garbage collection (GC) for automatic memory reclamation. While many parallel and concurrent algorithms for GC appear in the literature (e.g., [2, 3, 6, 9–12, 17, 24]), a highly parallel system may fail to scale well if the shape of the object-graph is not suitable for a parallel trace. For example, tracing a large linked-list is sequential in nature, and cannot run in parallel. Amdahl's law says that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. In this case, no matter how parallel the rest of the heap is traced, the traversal of a long linked-list would require the time it takes to traverse it sequentially. Such performance problems are bothersome because the details of the runtime should be abstracted for the developer, who should be able to use any data structure without the need to consider lower-level garbage collection performance.

It was noted in the past that deep and narrow data structures would be difficult to trace in parallel [6]. Siebert [21] examined heap depths of the SPECjvm98 [22] benchmarks in order to point out problems with parallel garbage collection. He found several problematic benchmarks in the suite. Recently, Barabash and Petrank [4] have extended this investigation to cover the DaCapo [5] benchmark suite. They also proposed the *idealized trace utilization* measure, a more accurate measure for detecting when heap shapes may hinder tracing scalability. Their study detected several benchmarks that manifest bad heap shapes. Finally, they proposed and investigated a couple of directions for solving the problem.

In this work we start by analyzing the problematic Java benchmarks in order to understand which data structures they employ and how they create the problematic heap shapes. We study the benchmarks that were found problematic in [4] and additionally, we also examine the newer DaCapo [5] version 9.12-bach and SPECjvm2008 [23] benchmark suites. From this study, it turns out that in all these benchmarks, the main reasons for deep heap shapes are linked-lists and queues.

A second contribution of this work is an attempt to ameliorate the problem by providing alternative library data structures that can be used instead of the data structures that

[*] This work was supported by the Israeli Science Foundation grant No. 283/10.

the problematic programs employ. To make this solution adequate for use with legacy code, we strive to impose minimal changes to the original program. The goal is to provide a designated (modified) library function that, when used instead of the original data structure, reduces the problem with no need for further modifications to the original program. In particular, we propose a new lock-free implementation of the queue data structure that enables more scalability of the garbage collector. The new data structure has hidden references that the program does not use but help the collector scale the tracing phase. The new queue was implemented and used with some of the problematic benchmarks to show that its overhead is low and it can improve the CPU utilization on highly parallel platforms.

The rest of the paper is structured as follows. Section 2 provides some background about garbage collection and heap shape analysis. In Section 3 we provide an analysis of common Java benchmarks. Section 4 introduces the modified queue, and Section 5 provides experimental results using the proposed queue. Finally, Section 6 describes related work, and Section 7 concludes the paper and lays out opportunities for further research.

Some material is omitted from this short submission. For example, the detailed heap shape measurements for all the benchmarks we looked at, and a liveness proof for the proposed data structure. Everything is available in an extended version of this paper [14].

2. Background

Garbage collectors trace the program's heap to determine which objects are accessible by the program, and which can be reclaimed. They trace the heap starting from a set of root objects, and identify live objects as objects reachable from the roots through object references.

The *depth* of a live object is defined as the length of the shortest path from a root to it. The depth of the heap graph is the maximum depth over all the live objects in the heap. A deeper graph might indicate a long sequential operation required by the GC, as an object at depth d will require at least d sequential dereference operations to reach. However, it is possible that a heap will contain enough objects at the maximum depth with enough paths to them so that the trace could still be done in parallel. For example, a k-core processor can trace k linked-lists in parallel with excellent utilization, even if the lists are very long.

A better measure for the scalability of a given heap's trace is the *Idealized Trace Utilization* proposed in [4]. It approximates processor utilization during a trace of the heap, assuming perfect load balancing and instant scanning of objects. The method also assumes a BFS scan, which is geared toward higher parallelism. More specifically, the idealized trace utilization, on a given heap shape with a given number of simulated processors, is computed by calculating the number of *cycles* it takes to scan the heap, such that in each

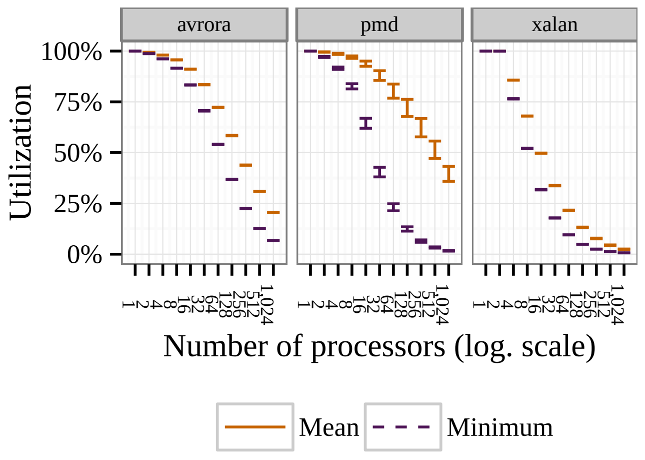

Figure 1: Idealized trace utilization, DaCapo 9.12 benchmarks

cycle, each processor takes a single object from a shared BFS queue, scans that object, and inserts all its non-scanned children to the end of the queue. The idealized trace utilization measure then is the total number of live objects in the heap, divided by the number of processors and the number of clock cycles. It can be used to evaluate the fraction of utilized CPU cycles at the best case in which the system has no load-balancing, cache-miss, or synchronization issues. When the utilization is low, it tells us that a parallel trace of the heap would have trouble scaling.

3. Benchmark Analysis

In [4] traces of several benchmarks of DaCapo 2006-10-MR2 and SPECjvm98 were examined by running them under the Jikes [1] Java Virtual Machine (JVM), modified to make frequent garbage collections and calculate the idealized trace utilization measure during these collections. We chose to calculate the idealized trace utilization differently, by running the benchmarks under Oracle$^{®}$ HotSpot^{TM} JVM and instrumenting the benchmarks to frequently write heap dumps to the disk. To do that we used Google's java-allocation-instrumenter [16], which is based on the ASM [7] bytecode manipulation library. We then calculated the idealized trace utilization on the heap dumps, to see which of the benchmarks would cause scalability problems for the garbage collector. This method allowed us to test a wider set of benchmarks than the ones reported in previous work [4, 21], including DaCapo 9.12 and SPECjvm2008, thus testing most if not all of the benchmarks commonly used for evaluating JVM garbage collectors.

Figures 1-2 show the idealized trace utilization for benchmarks of the later versions of the benchmark suites. We show only benchmarks that are relevant to this paper, i.e. benchmarks exhibiting significantly low utilization, or otherwise discussed in the paper. Each chart depicts the CPU utiliza-

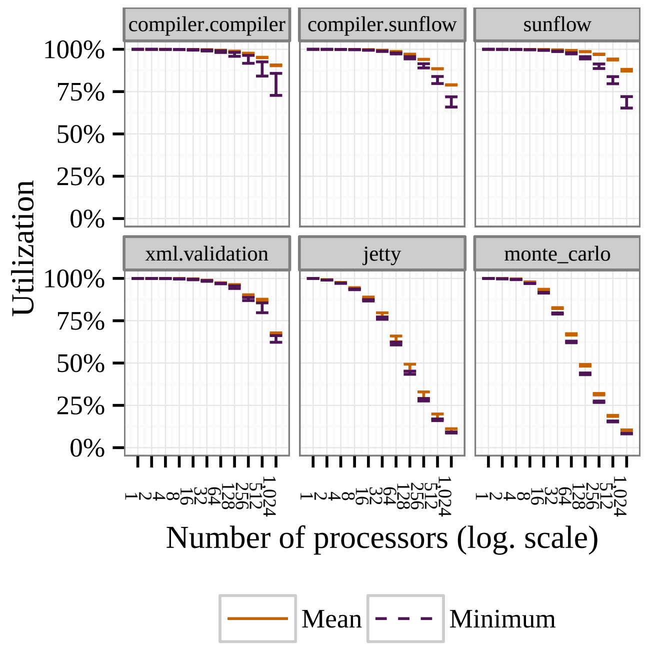

Figure 2: Idealized trace utilization, SPECjvm2008 benchmarks, and `Jetty`.

tion percentage as a function of the number of simulated processors. Benchmarks whose utilization is lower are suspected of causing GC scalability problems. The charts show confidence intervals for a confidence level of 95%, calculated using the Student's t-distribution. Note that the variance is many times so small that the confidence intervals collapse into something that looks like an empty interval.

While the utilization graphs allow us to find benchmarks with scalability problems, it is also worth noting the size of the heap also plays a role in determining whether a benchmark is problematic. For example, the `monte_carlo` benchmark from the SPECjvm2008 suite has low utilization, yet this follows from a small heap and a linked-list of a few hundred nodes, which do not cause a serious delay in tracing. Note also that the depth of the heap does not always imply low utilization. For example, the `xml.validation` benchmark from SPECjvm2008 has a deep structure of length over a thousand, but as can be seen from the idealized trace utilization measure, this benchmark does not create a major scalability problem since its heap is large and the parallel execution can find nodes to trace while the deep structure is being traversed by one core sequentially. So our target problematic benchmarks are those with low utilization and substantial heap size.

3.1 Cause of Non-Scalable Heap Shape

In this section we report our benchmark study. We looked at the benchmarks that exhibit low utilization with a large enough heap (with heap depths of more than 900 nodes), and analyzed the data structures they use. We investigated the way they use these data structures in order to find the cause of the problematic heap shape discovered during the execution. The common cause for all was an underlying linked-list, which was used as the base for different data structures, such as queues, hash-tables, or general use lists.

`pmd` The `pmd` benchmark of the DaCapo 2006 benchmark suite analyzes Java classes for source code problems. It uses the `javacc` parser generator, which uses custom linked-lists to keep a cache of parsed tokens. The version from DaCapo 2006 had low worst-case idealized trace utilization due to these tokens. The cache was kept only for a short duration and the average (or typical) shape of the heap during the execution behaved a lot better. In the DaCapo 9.12 version the benchmark uses multi-threading to work on multiple files simultaneously. While this allows the GC to trace the tokens lists of different threads simultaneously, the lengths of these lists varies between the threads, so the idealized trace utilization is still low in the worst case.

`xalan` The `xalan` benchmark, in both DaCapo 2006 and DaCapo 9.12, transforms XML files into HTML. The test harness program used in the DaCapo test suite creates a queue based on a linked-list, in order to distribute work to the threads participating in the benchmark. The queue typically has more than 8000 items at the beginning of the execution.

`javac / compiler` `javac` is the Java language compiler. The benchmark version of `javac` used in SPECjvm98 uses custom linked-lists to represent the bytecode instructions in a method. This list is long for some methods. The version used in SPECjvm2008 (the `compiler.compiler` and `compiler.sunflow` benchmarks) did not manifest similar problems.

`raytrace / mtrt` The `raytrace` and `mtrt` benchmarks from the SPECjvm98 suite perform ray-tracing. `raytrace` uses a single thread, while `mtrt` uses multiple threads. They use a custom linked-list temporarily when loading a scene to be rendered. The scene is loaded from disk to the linked-list, and then the list is traversed to create an Octree. In SPECjvm2008 these benchmarks were replaced by the `sunflow` benchmark (also available in DaCapo 9.12), which does not suffer from this problem.

`avrora` The `avrora` [25] benchmark from DaCapo 9.12, is a simulator for the AVR microcontroller, and for sensor networks based on AVR chips. It uses multiple threads for simulating the nodes of a sensor network in parallel, and uses a linked-list in order to synchronize between the different threads. In some executions, the list can become long, up to about 1900 nodes, causing a decrease in the idealized trace utilization measure.

3.2 More Benchmarks

As we focus on Java's concurrent queues, let us attempt to add more benchmarks that might be relevant for testing the queue that we propose later in Section 4 below. We need applications that make use of the queue and for which GC scalability problems might arise.

Jetty Jetty is a Java based web-server. One of its modules, the Quality of Service filter, limits the number of active requests to a fixed number, in order to control access to some limited resource. When the available slots for active requests are full, the incoming requests are held in a `java.util.concurrent.ConcurrentLinkedQueue`, Java's concurrent queue implementation, until either another request completes, or a timeout occurs.

In order to create long linked-lists and to present a GC scalability problem, we have created a benchmark using the quality of service filter. We set the limit on the number of incoming requests to the number of the server's processing cores, and the clients were set to create many (10,000) concurrent (trivial) requests to the server. The requests were eventually served by code that included a fixed time delay of 50ms, in order to simulate real request processing. The Apache ab HTTP benchmarking tool was used to create the requests and send them to the server. `Jetty`'s idealized trace utilization chart is shown in Figure 2.

3.3 An Artificial Benchmark

Finally, in addition to looking into real world applications as benchmarks, we also created an artificial benchmark in order to have direct control on the heap. Our benchmark employs Java's concurrent queue and attempts to keep it at some fixed length, which is given as a parameter, while inserting and removing elements by multiple threads. In order to do that, the benchmark also maintains an atomic counter that each thread increments or decrements when inserting or removing elements. The worker threads constantly read the counter, and decide whether to enqueue or dequeue an element based on whether the counter's value is higher or lower than the target queue length.

4. Shortcut Queue

The queue is the most appropriate candidate for improvement with respect to garbage collection. Parallel application developers may use a (possibly long) queue for producer-consumer scenarios, without knowing that it badly affects the runtime scalability. In this study we looked at Java, but the solution we present applies to other garbage collected languages as well. The Java library offers an implementation of a lock-free unbounded queue, the `java.util.concurrent.ConcurrentLinkedQueue` class, based on an algorithm by Michael and Scott [19]. The implementation of this queue is based on a linked-list, and thus when the queue size becomes large, the queue can adversely affect a parallel garbage collector's run time. In this section, we propose an extension of this data structure that will enable the garbage collector to perform better, while keeping the user interface, the efficiency, the lock-free property, and the simplicity of the original data structure. The liveness proof of the new

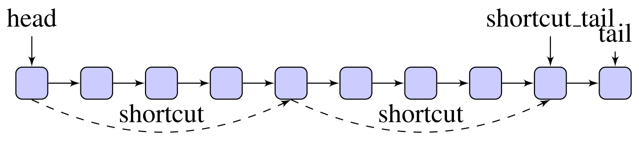

Figure 3: An illustration of a queue with shortcut references.

data structure is similar to the proof for the original queue in [19], and is omitted from this short submission. It is available in [14].

The new data structure adds shortcut references between nodes of the queue. Each node in the queue has a new field, named `shortcut`. These fields form an additional linked-list, which is designed to contain only every nth node (see Figure 3). These extra references allow the garbage collector to discover deeper parts of the queue early on, making more garbage collector threads participate in the trace. To keep record of the last node in this linked-list of shortcuts, a new shared `shortcut_tail` field was added to the queue. Due to arbitrary scheduling, shortcuts are not always created with the intended distance, but we found the effect of these irregularities to be insignificant. Further discussion of this phenomena can be found in [14].

4.1 The Enqueue Operation

The enqueue operation is depicted in Figure 4. When adding a new element to the queue, a check is first made to see if the enqueued node should be added to the shortcut list. Recall that only one in n nodes participates in this list. The check uses a shared counter to get an estimate of the number of elements enqueued so far. Whenever the count is divisible by n, a new shortcut is added. Note that since the threads do not update the counter atomically with an enqueue or a dequeue, the counter cannot be accurate when concurrent operations occur.

The thread creates a new node and attempts to insert it to the list by performing a Compare and Swap (CAS) operation on the `next` field of the last node in the list. In case of failure, the thread continues to try in a loop, until it succeeds.

After a node has been successfully inserted, the node is also inserted to the shortcut list. The code follows the same method of inserting a node to the queue's list. See Figure 5. It finds the last node on the shortcut list, starting from the `shortcut_tail` field of the queue, It then tries to perform a CAS operation on the `shortcut` field of the node. In case of failure, it continues attempting the insertion in a loop. Once it has succeeded inserting the node to the shortcut list, the code tries to update the queue's `shortcut_tail` reference to the new node.

Our changes to the original lock-free queue add two more CAS operations (in the `addShortcut` method) to the existing two CAS operations in the original enqueue algorithm. A third CAS operation is added when using the counter-based

```
enqueue(value) {
  node = new Node(value);
  boolean needsShortcut = needsShortcut();
  for (;;) {
    // Read the shared tail and next.
    t = tail; next = t.next;
    if (t != tail) continue;
    if (next == null) {
      // Tail points to last node.
      if (CAS(&t.next, null, node)) break;
    } else // Try to advance tail.
      CAS(&tail, t, next);
  }
  CAS(&tail, t, node); // Try to advance tail
  if (needsShortcut) addShortcut(node);
}
```

Figure 4: Enqueue operation pseudo-code. Differences from the original algorithm are highlighted.

```
addShortcut(node) {
  for (;;) {
    // Read the shortcut list's tail and next.
    sc = shortcut_tail; next = sc.shortcut;
    if (sc != shortcut_tail) continue;
    if (next == null) {
      // Tail points to last node
      if (CAS(&sc.shortcut, null, node))
        break;
    }
    else // Try to advance tail.
      CAS(&shortcut_tail, sc, next);
  }
  // Try to advance tail.
  CAS(&shortcut_tail, sc, node);
}
```

Figure 5: Pseudo-code for adding a node to the shortcut list.

method for deciding when to add new shortcuts. The two CAS operations used in the addShortcut method are only used when a new shortcut is needed, which happens infrequently and thus they do not add a significant overhead to the queue. As the measurements show, the additional CAS operation that is executed every time has a negligible overhead on the queue performance.

4.2 The Dequeue Operation

The dequeue operation is identical to the original version. A thread attempting to dequeue a node first reads the current values of head and tail. If they are equal, it checks whether head points at the last node, by checking if the node's next field is null. In this case, the queue is empty. Otherwise, the tail reference is lagging behind, and we attempt to advance it to next node. If the head and tail are different, the thread attempts to advance the head by a CAS operation, and if successful, it returns the value from the next node (the queue always contains a dummy node at the head). If the CAS fails, the thread restarts its operation.

Note that since the shortcut references are only intended to be used by the garbage collector, we do not need to maintain a "shortcut_head" reference, and we do not remove nodes from the shortcut list. Once a node is dequeued and removed from the list, it implicitly becomes garbage, and if it had a shortcut reference, then this reference will no longer be used by the garbage collector. This means that we do not add CAS instructions to the dequeue operation, or any operations at all, and its performance remains the same as the original queue.

5. Measurements

In order to check the benefits of the proposed new queue, we ran the benchmarks presented in Section 3, and computed the idealized trace utilization for the original benchmarks and for benchmarks modified to use the queue with shortcut references as proposed in Section 4. We present these results in Subsection 5.1. We also measured the time our benchmarks spent in the garbage collector, for benchmarks that use the original data structures and the modified ones. These measurements are discussed in Subsection 5.2. Finally, in Subsection 5.3 we also compared the performance of the benchmarks with and without our changes.

The experiments were run on an IBM x3400 server, featuring two Intel® Xeon® E5310 1.6 GHz quad core processors, and 16 GB of RAM. We used Oracle Java HotSpot 64-bit JVM version 1.7.0-b147, and IBM J9 VM 64-bit version 1.6 (SR9). The HotSpot JVM was run with the parallel scavenge garbage collector (-XX:+UseParallelGC), and the IBM JVM was run with the default throughput collector. We ran each benchmark multiple times in a single JVM instance, taking an average of the measured values over the multiple iterations, and ignoring the first iteration in order to focus on the steady-state results. We repeated each experiment 5 times, and we show the mean value over these runs, as well as confidence intervals for a confidence level of 95%, calculated using the Student's t-distribution. For each experiment that used a data structure with shortcuts, we picked the shortcut distance to be approximately a square root of the linked-list size, in order to get optimal results. When measuring the idealized trace utilization, we only used the HotSpot JVM, since this measure does not depend on the architecture, but on heap shape only, and also since our instrumentation mechanism was not compatible with the IBM JVM.

We compared our queue with shortcuts against the standard Java library's `ConcurrentLinkedQueue`, written by Doug Lea. This implementation is different from the Michael and Scott algorithm in that it uses garbage collection, so it does not need modification counters to avoid the ABA problem. Doug Lea's version also implements a few extra operations that are required by the `java.util.Queue` interface, such as removing an element from the middle of the queue, and iterating through the queue's elements without removing them. In addition, a couple of effective optimizations are implemented for this library queue implementation. First, it reduces the number of CAS operations, by letting the `head` and `tail` references be updated only every other operation and not at each enqueue or dequeue operation as in the original algorithm. Second, it sets the `next` reference of a node being removed to point to itself, to prevent a thread that loses its timeslice in a dequeue operation from forcing the garbage collector to keep more nodes alive unnecessarily. We added the same optimizations to our queue implementation, in order to obtain a fair comparison.

5.1 Idealized Trace Utilization

In Figure 6 the idealized trace utilization of the benchmarks described in Section 3 is compared with executions that make use of shortcuts. For the `xalan`, `Jetty` and `mtrt` we used the shortcut queue described in Section 4. For the `pmd` benchmark we implemented an ad-hoc list with shortcuts. The list used there is actually an extended queue that allows the program to keep references to nodes in the middle of the list, in addition to the head and tail references. It is easy to add shortcuts to such an enhanced queue building on the design of the shortcut queue of Section 4. Figure 6's charts are placed in a grid, each benchmark is shown with the mean idealized trace utilization results, next to a cell showing the minimum utilization results. Each chart shows the idealized trace utilization as a function of the number of simulated processors, for the original and for the modified data structures.

The `xalan`, `mtrt` and `Jetty` benchmarks showed a significant improvement of the trace utilization measure. Note that while sometimes the improvement in the utilization with 1024 simulated processors was not dramatic, there was always a drastic improvement with the lower number of processors. The `pmd` benchmark also improved its utilization, but the more significant improvement was with the minimal value, while the mean utilization did not improve as much. This is because the worst case heap shape in `pmd` occurred only in a small number of samples during each run, and so the mean value of the idealized trace utilization measure was higher to begin with.

Finally, we ran our artificial benchmark with different queue lengths, see Figure 7. Each chart in the row stands for a different queue length, showing the mean idealized trace utilization against the number of simulated processors. The figure shows how the scalability problem becomes more acute as the length of the queue grows, and how using shortcuts alleviates the problem.

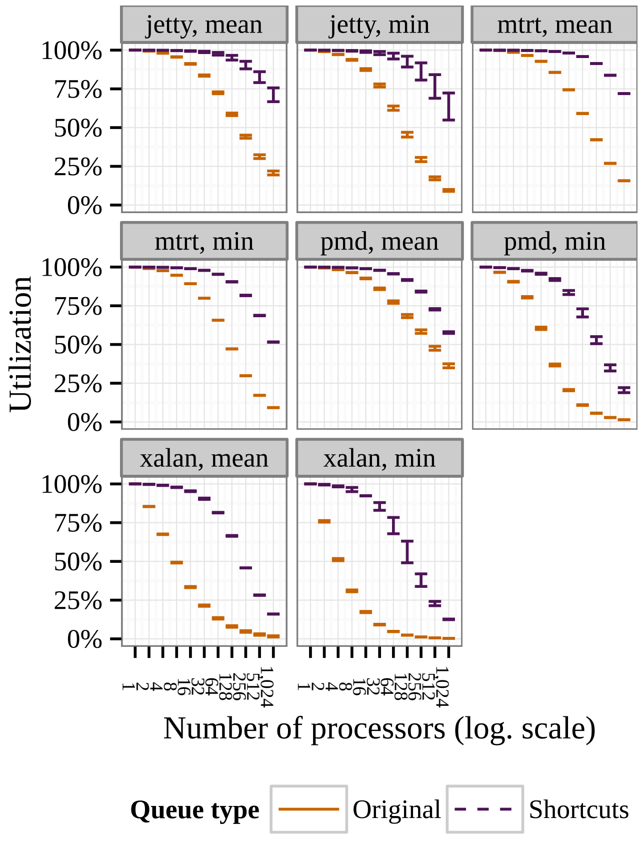

Figure 6: Idealized trace utilization for the original and modified benchmarks

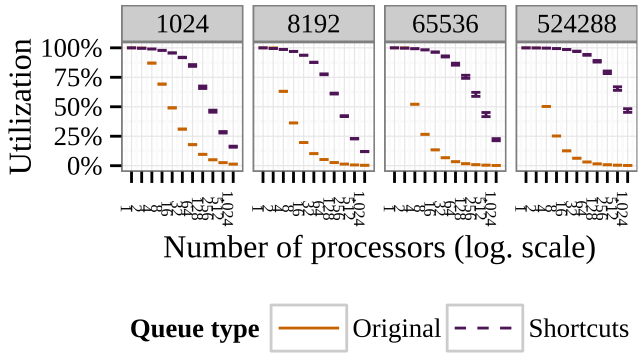

Figure 7: Idealized trace utilization for the artificial benchmark

5.2 Garbage Collection Time

Figure 8 shows the garbage collection time for executions of the artificial benchmark. Since this measurement is very sensitive to the garbage collection implementation (which we

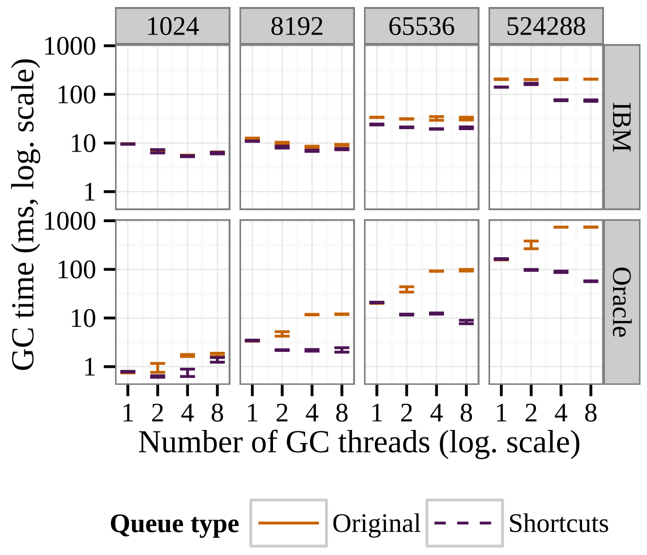

Figure 8: Artificial benchmark garbage collection time

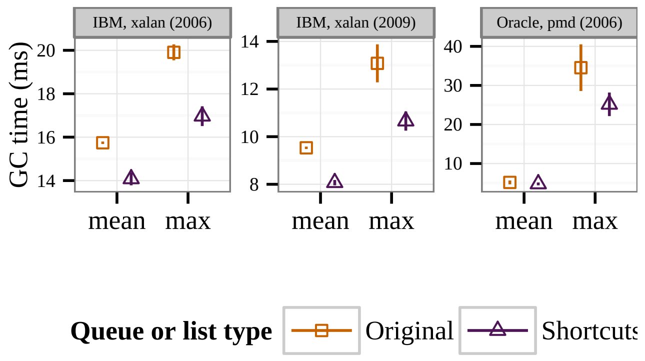

Figure 9: GC pause time comparing original and modified benchmarks. Showing only benchmarks whose running time were significantly different.

do not control), we checked the times both for the HotSpot JVM and for IBM's JVM. The chart shows in each column a different target queue length, with the first row showing results in IBM's JVM and the second row in Oracle's HotSpot JVM. Each cell in the chart shows the mean running time of the GC as a function of the number of GC threads used.

With the HotSpot JVM and without the shortcuts, adding more GC threads does not improve performance. In fact, performance deteriorates in this case. We discovered that this happens because of the contention on the JVM's stealing queues mechanism [15], when no parallel work is available for distribution among the GC threads. When using the shortcut queues, this problem disappears, and thus the speedup when using the shortcuts version can sometimes exceed the number of cores in our machine (8), and the effect of the modified queues is noticeable even with relatively short queues.

The IBM JVM uses a more coarse load-balancing algorithm [18], and does not suffer from the contention problem in the same magnitude. Therefore, in the runs on the IBM JVM, we notice the benefit of the modified queues only with larger queues.

We also compared the GC pause times for the other benchmarks using the enhanced data structures, under HotSpot and IBM's JVMs (Figure 9). While the idealized trace utilization measure can show benefits for a large number of threads, our actual machine is limited to eight cores, which are not enough to make the difference in performance visible for these queue lengths. For almost all the benchmarks, there was no significant difference between the original versions and the modified versions. The `pmd` benchmark of DaCapo 2006 is the exception to the rule, as it uses lists that are long enough to make a difference on eight machines as well. Running under the HotSpot JVM its maximum GC time was 29% shorter with the shortcuts. Under IBM's JVM, the `xalan` benchmark showed an improvement, running between 10%–20% faster in both the mean and the maximum collections.

5.3 General Performance

We examined the overall performance of runs with the original benchmarks and runs that use the enhanced data structures, under HotSpot and IBM's JVMs. In almost all tests, the difference in performance was not visible. Indeed, we did not expect to see a larger effect than that of GC time only, but these results show that the overhead of maintaining the shortcuts in the queue is negligible.

We also compared the performance of the artificial queue benchmark with and without the shortcuts on our eight-core machine. Under both JVMs, there were no significant differences in performance, except when running the longest queue length under the HotSpot JVM, in which case the modified version was more than twice as fast. This is probably due to the load balancing contention mentioned in Subsection 5.2.

6. Related Work

Most of the relevant related work (and especially [4, 21]) is already discussed in the introduction. More relevant work is discussed in this section. Endo et al. [13] developed a model for predicting parallel garbage collection performance while executing the program sequentially. They take into account several issues related to cache misses, load balancing, and the size and depth of the object graph. Raman et al. [20] propose a method for speculative parallelization of legacy code that dynamically maintain additional pointers to a data structure. Similarly to our shortcut references, these pointers can aid the garbage collector in parallel tracing. Since they work dynamically, their method depends on the user-code to do full traversals of the data structures, before it can start keeping any additional pointers, while we maintain the additional pointers throughout all the data structures' operations. Click [8] proposed using idle processors to start tracing random heap objects, aiding the trace of non-scalable heaps. This idea was partially evaluated in [4].

7. Conclusion

The problem of heap shapes that foil GC scalability has been raised and measured in previous work. In this paper we extended the measurements to cover additional benchmarks and we investigated the benchmarks that manifested such problems. We discovered that the main issue is with parallel programs employing the linked-list or the queue data structures. We then proposed a new design of an enhanced lock-free queue that does not cause scalability problems for the collector. We have modified the benchmarks to use the enhanced data structures and measured the resulting executions. With the modified benchmarks, the heap shape was dramatically improved by employing our new shortcut queue, while not posing noticeable overhead on benchmark performance.

There are several directions worth further investigation. First, it would be interesting to have an automated tool that, given a program, checks whether it has GC scalability problems, and if so, which data-structures (or Java classes) are involved in creating the problem. Another direction is to try and make the garbage collection solve such problems automatically by installing invisible pointers in one collection cycle, that may be used by the next collection cycles.

References

[1] Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The jikes research virtual machine project: building an open-source research community. IBM Systems Journal 44(2)

[2] Azatchi, H., Levanoni, Y., Paz, H., Petrank, E.: An on-the-fly mark and sweep garbage collector based on sliding views. In: OOPSLA. pp. 269–281. ACM (2003)

[3] Barabash, K., Ben-Yitzhak, O., Goft, I., Kolodner, E.K., Leikehman, V., Ossia, Y., Owshanko, A., Petrank, E.: A parallel, incremental, mostly concurrent garbage collector for servers. TOPLAS 27(6), 1097–1146 (Nov 2005)

[4] Barabash, K., Petrank, E.: Tracing garbage collection on highly parallel platforms. In: ISMM '10. ACM (Jun 2010)

[5] Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA. pp. 169–190. ACM (2006)

[6] Boehm, H.J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. In: PLDI '91. pp. 157–164. ACM (1991)

[7] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: In Adaptable and extensible component systems. Grenoble, France (2002)

[8] Click, C.: Private communication

[9] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. CACM 21(11), 965–975 (Nov 1978)

[10] Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: POPL '94. ACM

[11] Domani, T., Kolodner, E.K., Petrank, E.: A generational on-the-fly garbage collector for Java. In: PLDI '00. ACM (2000)

[12] Endo, T., Taura, K., Yonezawa, A.: A scalable mark-sweep garbage collector on large-scale shared-memory machines. In: ICS (Nov 1997)

[13] Endo, T., Taura, K., Yonezawa, A.: Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In: IPDPS '01. pp. 43–48 (2001)

[14] Eran, H.: A Study of Data Structures with a Deep Heap Shape. Master's thesis, Computer Science Department, Technion – Israel Institute of Technology (Apr 2012), `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2012/MSC/MSC-2012-13.pdf`

[15] Flood, C., Detlefs, D., Shavit, N., Zhang, C.: Parallel garbage collection for shared memory multiprocessors. In: JVM '01. USENIX, Monterey, CA (Apr 2001)

[16] Google Inc.: java-allocation-instrumenter: A Java agent that rewrites bytecode to instrument allocation sites (2009), `http://code.google.com/p/java-allocation-instrumenter/`

[17] Hudson, R.L., Moss, J.E.B.: Sapphire: Copying GC without stopping the world. In: Java Grande. ACM (2001)

[18] IBM: IBM SDK and runtime environment Java technology edition version 6, `http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=%2Fcom.ibm.java.doc.diagnostics.60%2Fdiag%2Funderstanding%2Fmm_gc_mark.html`

[19] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC '96. pp. 267–275. ACM (1996)

[20] Raman, E., Vachharajani, N., Rangan, R., August, D.I.: Spice: speculative parallel iteration chunk execution. In: CGO '08. pp. 175–184. ACM (2008)

[21] Siebert, F.: Limits of parallel marking collection. In: ISMM '08. pp. 21–29. ACM, Tucson, AZ (Jun 2008)

[22] Standard Performance Evaluation Corporation (SPEC): SPECjvm98 (1998), `http://www.spec.org/jvm98`

[23] Standard Performance Evaluation Corporation (SPEC): SPECjvm2008 (2008), `http://www.spec.org/jvm2008`

[24] Steele, G.L.: Multiprocessing compactifying garbage collection. CACM 18(9), 495–508 (Sep 1975)

[25] Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: IPSN '05. IEEE