A Study of Data Structures with a Deep Heap Shape

Haggai Eran

Technion - Computer Science Department - M.Sc. Thesis MSC-2012-13 - 2012

A Study of Data Structures with a Deep Heap Shape

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Haggai Eran

Submitted to the Senate of
the Technion — Israel Institute of Technology
Iyar 5772 Haifa April 2012

The research thesis was done under the supervision of Associate Professor Erez Petrank in the Computer Science Department.

I would like to sincerely thank my advisor, Assoc. Prof. Erez Petrank, for his guidance. During our work together he both challenged me to do better, and bestowed confidence in my abilities. His optimistic point of view helped me to continue when things didn't work out as well as we had hoped.

I wish to thank my thesis examiners, Assoc. Prof. Idit Keidar and Dr. Eran Yahav, and the anonymous conference reviewers who reviewed our work. Their comments and questions enabled us to improve the quality of this work.

I also want to thank my family and my community, for their encouragement and support.

Finally, I want to thank my wife Shoshan, for her love and her understanding during this period, and for showing her interest and participating in this part of my life.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

List of Tables

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. Such structures can hurt the load balancing of the parallel trace, and that may hinder parallel tracing. In this work we start by analyzing which data structures make current Java benchmarks create deep heap shapes. We analyze benchmarks of four common benchmark suites, measuring how well they might scale under a parallel garbage collector. We use a simulated multiprocessor garbage collection trace for our measurements, allowing us to estimate the scalability of the garbage collector on massively parallel machines.

It turns out that the problem is manifested mostly with benchmarks that employ queues and linked-lists. We discuss existing alternatives for these data structures, that would improve the garbage collector scalability. Then we propose a new construction of the queue data structure that enables better garbage collector parallelism. We modify an existing lock-free unbounded queue by adding extra references that allow the garbage collector to distribute the tracing of the queue among multiple threads. The new queue incurs a low overhead, and provides the same progress guarantees as the original queue.

Abbreviations and Notations

BFS	Breadth First Search
CAS	Compare and Swap
GC	Garbage Collection / Collector
HTML	Hyper-text Markup Language
HTTP	Hyper-text Transfer Protocol
JVM	Java Virtual Machine
RAM	Random Access Memory
XML	Extensible Markup Language
l	List length: the length of a linked-list or a queue
n	Shortcut distance: the number of nodes between shortcuts
p	Shortcut probability: the probability of inserting a shortcut to the queue

Chapter 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™ and C# use garbage collection (GC) for automatic memory reclamation. While many parallel and concurrent algorithms for GC appear in the literature [34, 35, 11, 7, 13, 12, 17, 15, 16, 27, 9, 22, 24, 3, 19, 2, 4], 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 developers may not know that their use of a specific data structure makes parts of the runtime scale badly, and they do not understand why.

It was noted long ago [7] that deep and narrow data structures would be difficult to trace in parallel. Siebert [31] examined heap depths of the SPECjvm98 [32] benchmarks in order to point out problems with parallel garbage collection. He found several problematic benchmarks in the suite. Recently, Barabash and Petrank [5] have extended this investigation to a more comprehensive study of Java benchmarks. They proposed the *ideal-*

ized 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 investigate 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 [5] and also studied the DaCapo [6] version 9.12-bach and SPECjvm2008 [33] benchmark suites. From this study, it turns out that the main reason for deep heap shapes in these benchmarks are linked-lists and queues.

A second contribution of this work is an attempt to ameliorate the problem by providing alternative data structures that can be used instead of the data structures that 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 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 processor utilization on highly parallel platforms. In addition to the queue, we also examined the use of skip-lists [28] to replace linked-lists, as a more scalable alternative.

1.1 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 all live objects as those reachable from the roots through object references.

As live objects are those that have a path from a root to the object, an object's *depth* is defined as the length of the shortest such path. The depth of the heap graph is the maximum depth of 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* measure that was proposed in [5]. It approximates processor utilization during a trace of the heap, assuming perfect load balancing and instant scanning of objects. The method also assumes a Breadth First Search (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 cycle, each processor takes a single object from a shared BFS queue, scans that object, and inserts all its non-scanned children to 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 processor 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.

1.2 Related Work

Most of the relevant related work (and especially [31, 5]) is already discussed in this chapter. More relevant work is discussed in this section. Endo et al. [18] 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. [29] 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 [10] proposed using idle processors to start tracing random heap objects, aiding the trace of non-scalable heaps. This idea was partially evaluated in [5].

Chapter 2

Benchmark Analysis

In [5] 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™ JVM and instrumenting the benchmarks to frequently write heap dumps to the disk. To do that we used Google's java-allocation-instrumenter [21], which is based on the ASM [8] 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 [31, 5], including DaCapo 9.12 and SPECjvm2008, thus testing most if not all of the benchmarks commonly used for evaluating JVM garbage collectors. Using a different JVM and class library naturally caused some differences in the contents of the heap. We excluded such differences in our report, so that our results are comparable with the previous work. In chapter 6 we provide the full results.

Figures 2.1, 2.2, 2.3, and 2.4 show the idealized trace utilization for benchmarks of each of these benchmark suites. Each chart depicts the processor utilization 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

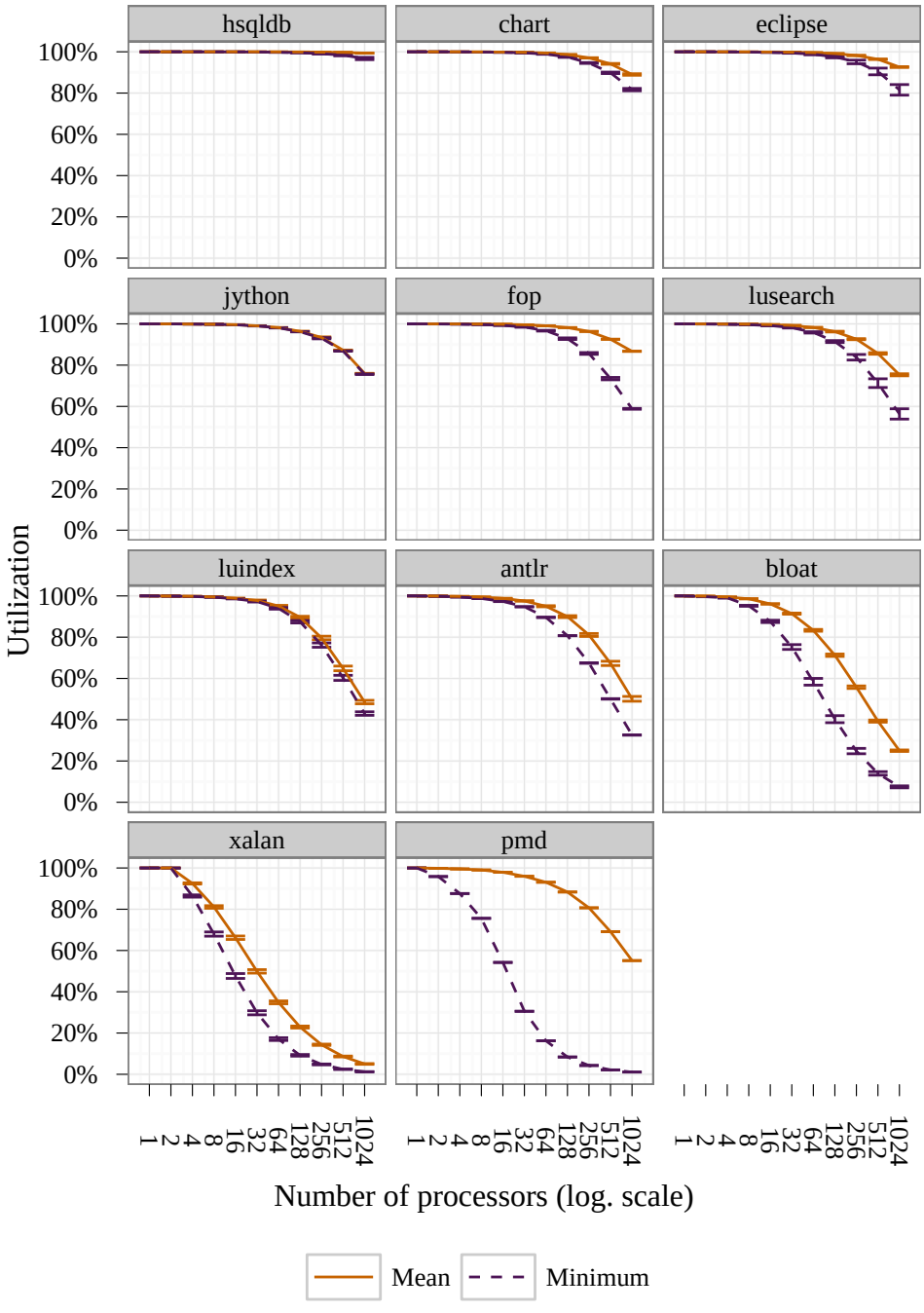

Figure 2.1: Idealized trace utilization, DaCapo 2006 benchmarks

Benchmark	Mean depth	Max depth	Heap-dumps
pmd	360.2	28599.5	147.5
xalan	4237.5	8399.4	42.9
bloat	351.9	733.6	49.5
eclipse	45.3	80.7	34.4
antlr	28.3	55.0	29.2
hsqldb	29.4	45.0	11.5
jython	39.0	39.0	23.2
fop	27.8	29.4	27.8
lusearch	26.8	26.8	23.2
luindex	25.6	25.6	12.5
chart	25.4	25.4	7.5

Table 2.1: Maximum depth of an object in DaCapo 2006

Benchmark	Mean depth	Max depth	Heap-dumps
pmd	4104.8	33587.6	17.3
xalan	4257.5	8432.3	66.1
avrora	985.0	1907.8	19.8
fop	462.9	868.5	10.5
eclipse	439.3	558.0	5.0
batik	262.4	340.2	77.8
tradebeans	211.7	274.1	6.8
tradesoap	199.4	264.9	12.8
jython	86.0	86.0	9.8
tomcat	45.1	54.0	9.2
h2	31.1	45.9	12.7
sunflow	27.0	31.8	14.2
luindex	26.2	26.2	6.5
lusearch	25.8	25.8	139.1

Table 2.2: Maximum depth of an object in DaCapo 9.12

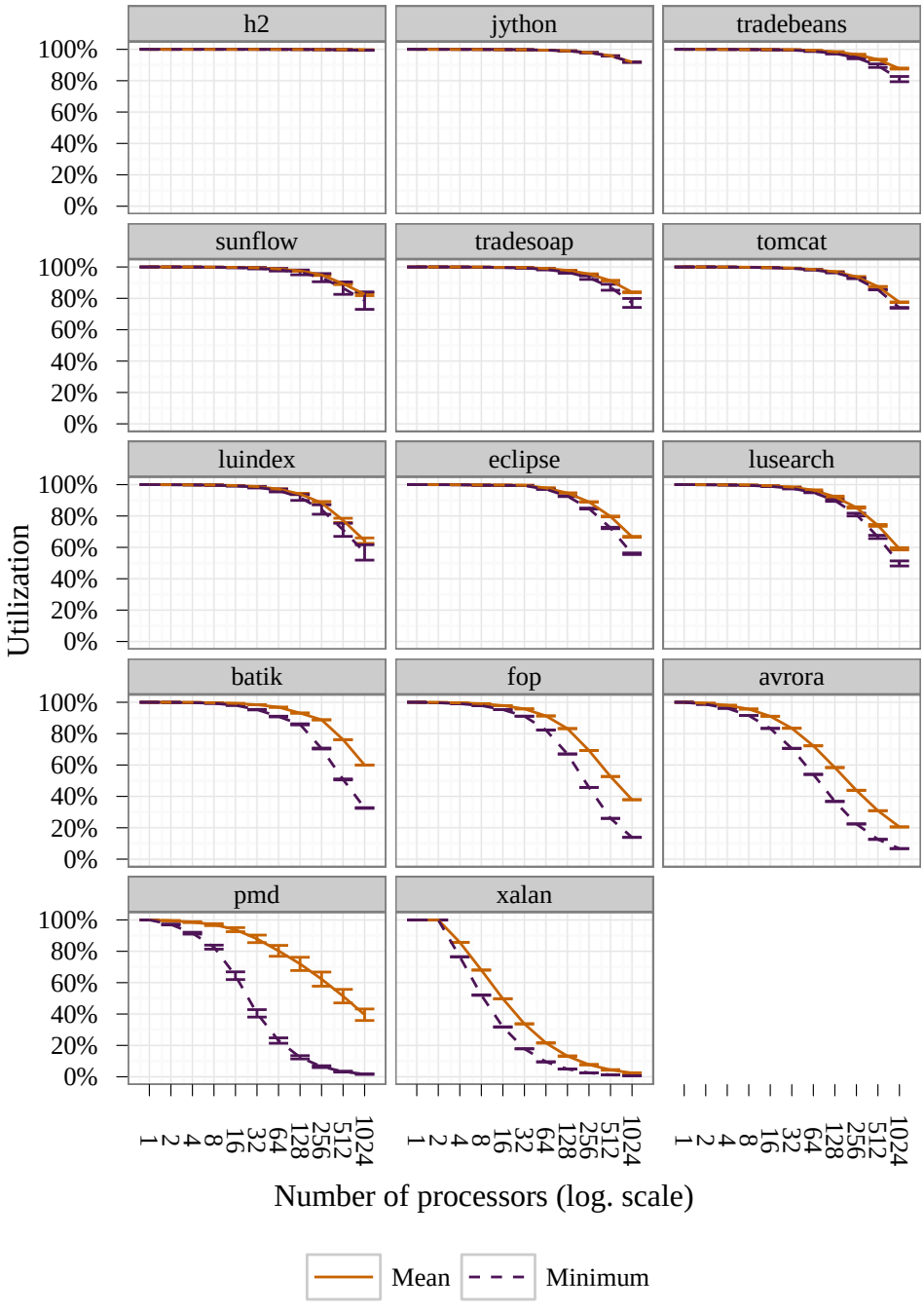

Figure 2.2: Idealized trace utilization, Dacapo 9.12 benchmarks

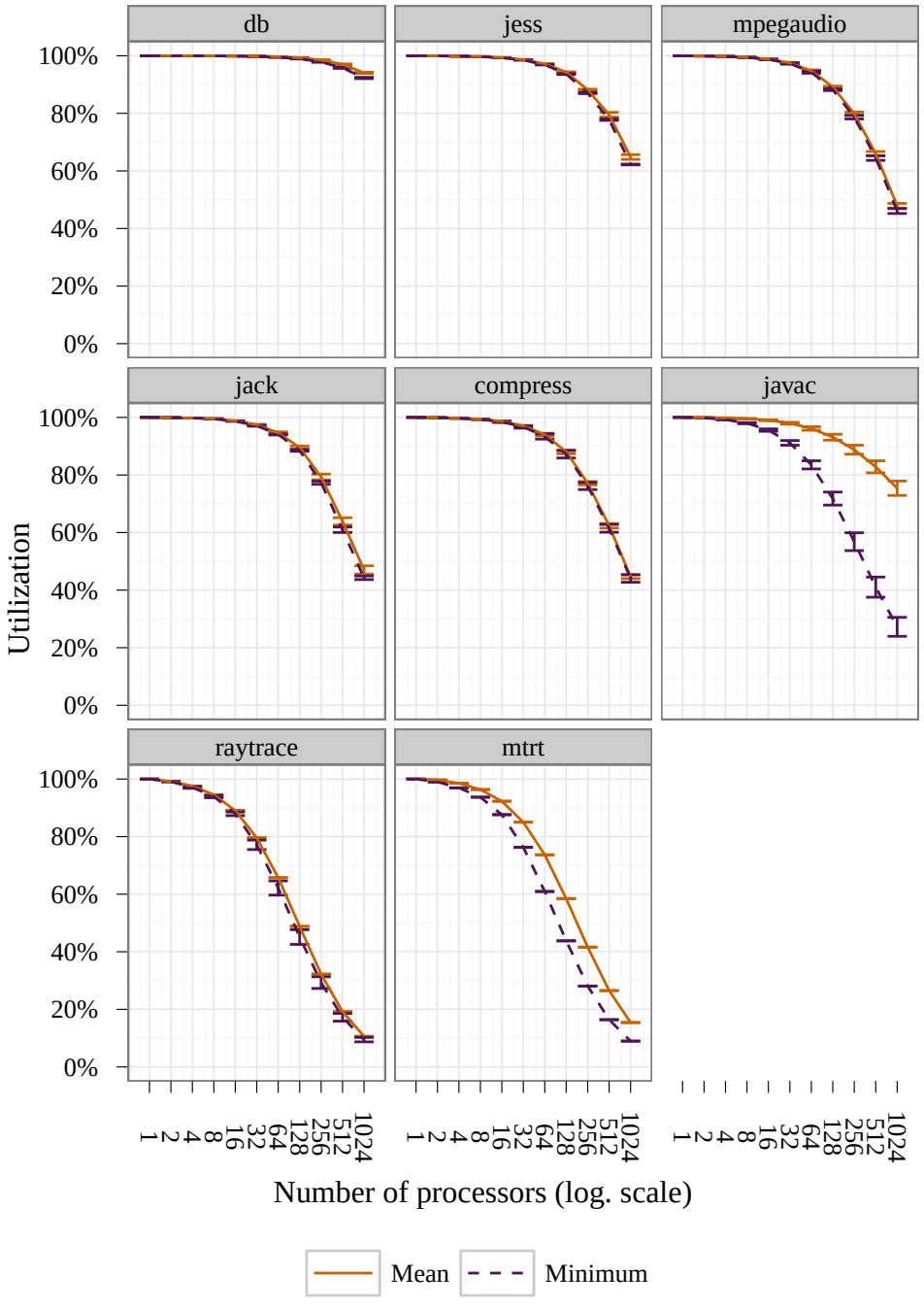

Figure 2.3: Idealized trace utilization, SPECjvm98 benchmarks

Benchmark	Mean depth	Max depth	Heap-dumps
mtrt	1405.7	1413.0	11.3
raytrace	1405.0	1413.0	6.1
javac	235.5	986.0	24.1
jack	34.3	34.8	5.5
jess	29.0	29.0	5.3
compress	26.2	27.6	133.5
mpegaudio	26.4	26.4	12.0
db	25.6	25.6	11.2

Table 2.3: Maximum depth of an object in SPECjvm98

Benchmark	Mean depth	Max depth	Heap-dumps
xml.validation	1180.9	1184.0	39.4
compiler.sunflow	563.0	669.0	53.1
compiler.compiler	429.7	519.9	30.7
crypto.rsa	424.7	512.9	92.6
crypto.signverify	390.9	442.2	223.3
xml.transform	258.0	281.6	96.7
monte_carlo	212.2	240.1	14.2
serial	146.8	171.5	45.8
crypto.aes	61.7	75.5	202.7
scimark.sor	41.8	65.3	13.9
scimark.fft	33.6	61.7	15.9
sunflow	25.9	51.0	122.2
scimark.lu	27.9	49.0	11.3
compress	26.0	26.0	13.0
scimark.sparse	25.9	25.9	13.1
derby	25.8	25.8	18.8
mpegaudio	25.5	25.5	8.0

Table 2.4: Maximum depth of an object in SPECjvm2008

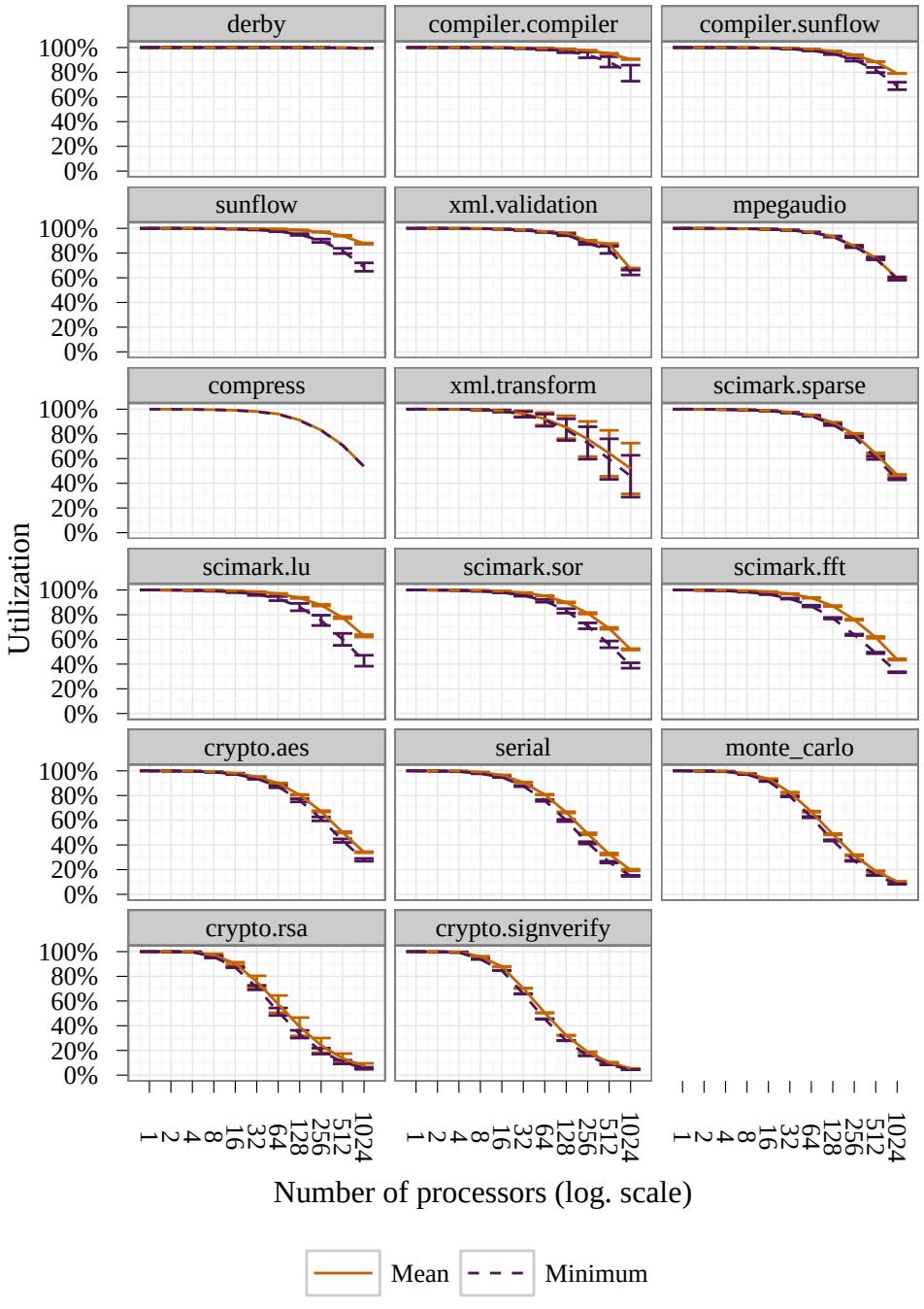

Figure 2.4: Idealized trace utilization, SPECjvm2008 benchmarks

problems, it is also worth noting the maximum depth of the object graph of each benchmarks, as it can be used as a lower bound on the number of object scans required to complete a garbage collection trace. Tables 2.1, 2.2, 2.3, and 2.4 shows the maximum depth in our set of benchmarks. For each benchmark it specifies the mean and the maximum value, and also the number of heap dumps taken in each iteration of the benchmark. It is interesting to note that a deep heap doesn't always come with low utilization measure, and vice versa. For example, that 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 don't cause a serious delay in tracing. In contrast, 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 larger and the parallel execution can find nodes to trace while the deep structure is being traversed.

In Figure 2.5 we show the results of our measurements for the four benchmarks suites. Both the idealized trace utilization (measured for 1024 simulated processors), and the maximum heap depth are shown. We have decided to focus on benchmarks that were showing low utilization, and a large and deep heap, which are marked in the chart with a black rectangle. The benchmark `bloat` was also checked, despite having a smaller heap, since it was mentioned in [5] as a benchmark that could cause problems for a parallel GC.

2.1 Cause of Non-Scalable Heap Shape

We have studied the benchmarks that exhibit low utilization with a large enough heap, 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 was always an underlying linked-list, which was used as the base for different data structures, such as queues, hash-tables, or general use lists. We now go over the benchmarks that manifest heaps with scalability problems.

`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, when the CACHE_TOKENS feature is enabled. The version from DaCapo 2006 had low worst-case idealized trace utilization due to these tokens. The cache was

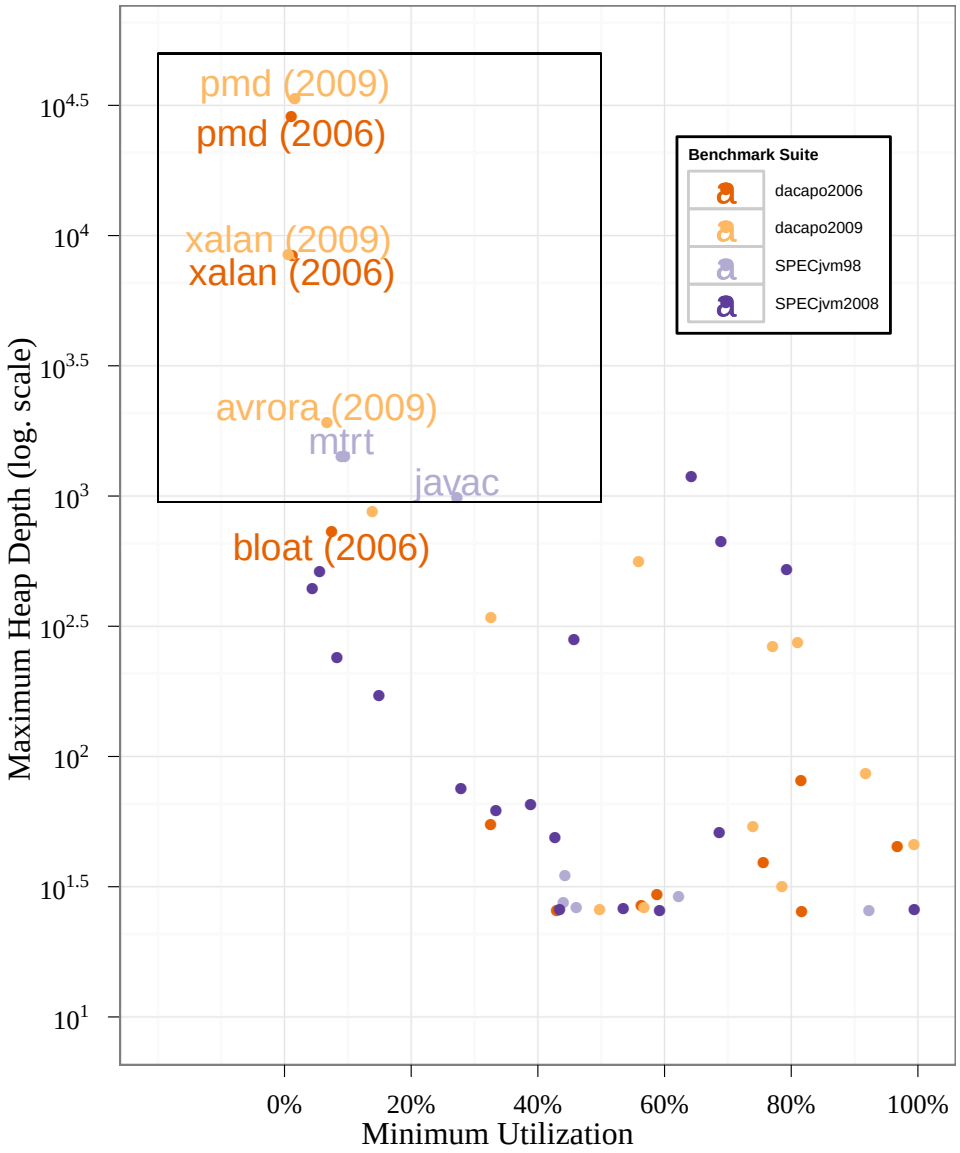

Figure 2.5: Idealized trace utilization (1024 processors) and maximum heap depth for benchmarks of the suites DaCapo 2006, DaCapo 9.12, SPECjvm98, and SPECjvm2008. Benchmarks labels are shown for the benchmarks that were analyzed in detail.

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) didn't manifest similar problems. The `javac` benchmark has changed in many ways between SPECjvm98 and SPECjvm2008. In its later form it uses immutable linked-lists to keep the bytecode instructions, and it uses a different set of programs as input.

In section 4.2 we propose new data structures that could replace such immutable linked-lists, and allow garbage collector scalability by adding shortcut references for the garbage collector use. We did not implement linked-list related data structures as it was not clear that one such implementation would be useful for various benchmarks.

`bloat` The `bloat` benchmark, from DaCapo 2006, performs optimization on the Java bytecode. Similarly to `javac`, it keeps instructions in linked-lists. However, it differs in that it uses the standard Java library's `java.util.LinkedList` to do that. In addition, `bloat` uses a hash table with a specifically non-uniform hash function, to implement one of its algorithms. The hash table uses linked-lists to keep conflicting entries. The non-uniformity of the hash function made a few of the collision lists become long, which also contributed to the benchmark's low idealized trace utilization.

`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` [37] 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. Each node in the list contains a time, and the list is ordered by these times. It also contains a counter that provides the number of threads that have already reached this time. When a thread decides it needs to wait for other threads to reach a certain time, it finds the right node in the list, creating it if necessary, and waits for the counter to reach the number of threads. In some executions, the list can become long, up to about 1900 nodes, causing a decrease in the idealized trace utilization measure. All access to this data structure is done while the accessing thread is holding a lock. The same lock is used by the condition variable that the threads use for waiting.

2.2 Scaling with Linked-Lists and Queues

We examine the data structures used by the problematic benchmarks and ask how can we modify them to make tracing collectors scale well. We first claim that there is no real potential for improvement with the serial linked-list data structure and we then focus our attention on modifying the queue data structure to make it adequate for GC environment on a highly parallel platform. Although the queue also builds on an underlying linked-list, it makes limited use of it in a way that allows an easier solution.

Except for `xalan` and `avrora`, all of the examples listed above make use of a linked-list that inherently assumes a single thread. These are all single-threaded application to start with, but they make special use of linked-lists that relies on sequential execution. In particular, they refer to objects in the linked-list according to their ordinal number or their position. In fact, the purpose of the list is to give order to elements in it, e.g. to represent the order of the byte codes in a routine. In such applications, e.g. `javac` and `bloat`, it is not clear how we can even define meaningful semantics for concurrent operations. What does it mean that several threads are trying to erase the fifth element? Are they all trying to erase the same element and all but one should realize the concurrent execution and fail? Or does it mean that whatever the fifth element is when they traverse the list should be deleted? In the sequential case, the fifth element is different after each

deletion, and applying such a delete several times would imply deleting several different such elements from the list. But in the parallel world the identity of the fifth element depends on which operations linearize prior to the delete's search procedure. This concurrent handling becomes even more involved when concurrent insertions are used as well. The `java.util.List` interface used by `bloat` and the custom linked-list used by `javac` allow the user to access, modify, and delete elements of the list, based on their position in the list. Applying the same semantics in the parallel world is not possible.

We could deal with such data structures by assuming that they are used by a single thread. However, this case is not very interesting, as we want to improve data structures that support parallel computation and let applications scale. If the application is run with no concurrency, then one would only gain a very small benefit by making the GC parallel, while the application itself runs on one of the many cores. Another possible scenario is when the application is parallel but each core works with a different list. In this case the tracing problem does not arise, as there are many lists to traverse and scaling does not become an issue. Furthermore, other solutions such as thread local heaps [14, 36, 25] can be used to deal with such cases. In spite of these benchmarks being imperfect for a parallelization study, we nevertheless attempted to replace `bloat`'s linear list with a specially designated linear *skip-list* to get a feeling on how such implementations could be improved. See section 4.1 for the details.

The other use of linked-lists is for key-sorted elements. Such lists are typically used to maintain sets or maps, and these linked-lists can also support parallelism, because concurrent searches, inserts and deletes (by key) are well defined. However, such uses of linked-lists can be easily modified to employ a *skip-list* instead of the linked-list. A skip-list would make the application run faster and the resulting heap shapes let the collector scale very well. So here, again, there is no use in proposing support for scaling the collection of the linked-list.

We therefore (except for the study of `bloat`) focus our attention on the queue. Replacing queues with skip-lists would incur noticeable overheads, as queues are only accessed at their ends and the skip-list overhead is never offset by fast access to the middle of the list. Furthermore, queues can be modified to support the scaling of tracing collectors with only a small overhead, as explained in chapter 3 below.

The `avrora` benchmark discussed above uses a priority queue in which items are always dequeued from one end but can be added in the middle according to a priority key. This benchmark employs an ad-hoc data structure for this purpose with coarse-grained locking. This data structure can

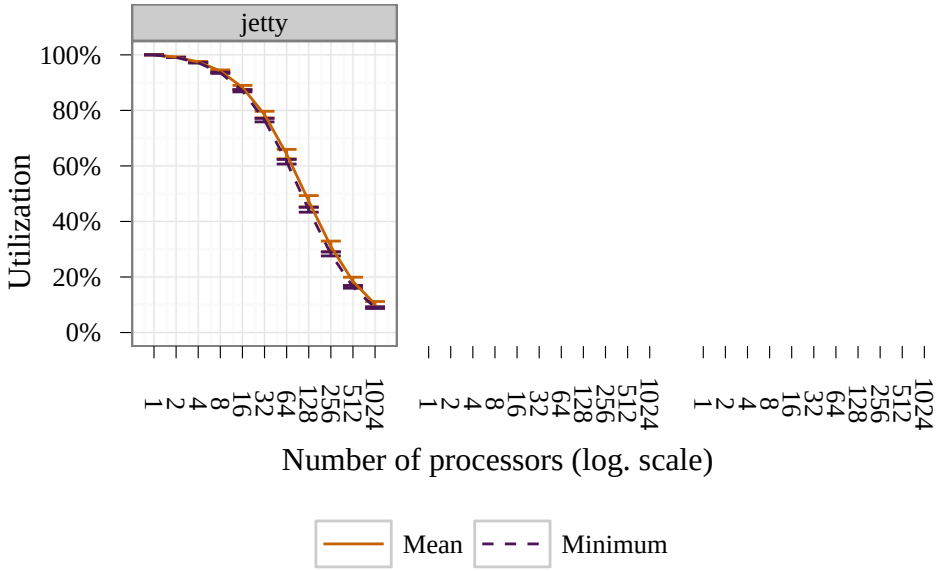

Figure 2.6: Idealized trace utilization, `Jetty` benchmarks

create a bad heap shape, but it can be easily replaced by a skip-list. (For this specific benchmark, one can even use a non-concurrent skip-list as all accesses to the extended queue are protected by a single lock.) As the use of the data structure is ad-hoc, we did not implement a special skip-list for it and did not take measurements. Our goal was to look for solutions for applications that use library methods.

Another alternative data structure for sets and maps is the hash table. Assuming uniform distribution and a large enough table, a hash table that uses chaining for conflict resolution wouldn't create a deep shape, and will allow the GC to scale. Shalev and Shavit [30] offered an extensible lock-free hash table data structure, implemented using one linked list. The hash-table array contain shortcut pointers to the hash buckets, which are just segments of the linked list. These pointers can also be used by the GC to divide the trace of the list more efficiently.

2.3 More Queue 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 chapter 3 below. We need applications that make use of the queue and for which GC scalability problems might arise.

Benchmark	Mean depth	Max depth	Heap-dumps
jetty	2666.7	3495.0	33.8

Table 2.5: Maximum depth of an object in `Jetty`

`Jetty` `Jetty` is a Java based web-server. It supports Servlet filters for performing various operations on requests and responses, and one of the included filters uses `java.util.concurrent.ConcurrentLinkedQueue`, Java's concurrent queue implementation which is based on a linked-list. 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 queue 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 heap depth is shown in Table 2.5, and its idealized trace utilization chart is shown in Figure 2.6.

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.

Chapter 3

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 [26]. 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 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 3.1). These extra references allow the garbage collector to discover deeper parts of the queue early on, making more garbage collector

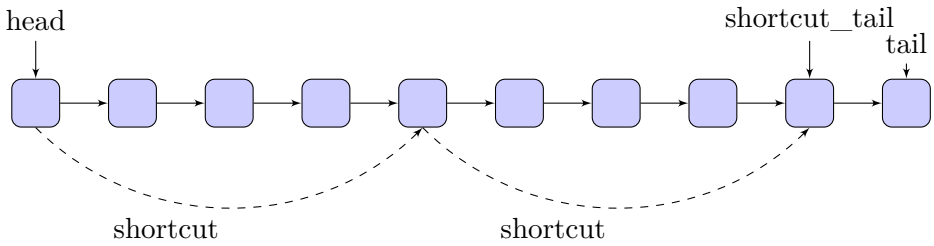

Figure 3.1: An illustration of a queue with shortcut references.

```
1  enqueue(value) {
2    node = new Node(value);
3    boolean needsShortcut = needsShortcut();
4    for (;;) {
5      // Read the shared tail and next.
6      t = tail; next = t.next;
7      if (t != tail) continue;
8      if (next == null) {
9        // Tail points to last node
10       if (CAS(&t.next, null, node)) break;
11     }
12     else
13       CAS(&tail, t, next); // Try to advance tail.
14   }
15   CAS(&tail, t, node); // Try to advance tail.
16   if (needsShortcut) addShortcut(node);
17 }
```

Figure 3.2: Enqueue operation pseudo-code. Differences from the original algorithm are highlighted.

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.

3.1 The Queue Operations

3.1.1 The Enqueue Operation

The enqueue operation is depicted in Figure 3.2. 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. We have created two implementations of this check. The first method 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 second method uses a random number generator and adds a shortcut with probability $p = 1/n$.

The thread creates a new node and attempts to insert it to the list by

```
1  addShortcut(node) {
2    for (;;) {
3      // Read the shortcut list's tail and next.
4      sc = shortcut_tail; next = sc.shortcut;
5      if (sc != shortcut_tail) continue;
6      if (next == null) {
7        // Tail points to last node
8        if (CAS(&sc.shortcut, null, node)) break;
9      }
10     else // Try to advance tail.
11       CAS(&shortcut_tail, sc, next);
12   }
13   // Try to advance tail.
14   CAS(&shortcut_tail, sc, node);
15 }
```

Figure 3.3: Pseudo-code for adding a node to the shortcut list.

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 3.3. 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 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.

```
1  dequeue() {
2    for (;;) {
3      // Read the shared head and tail.
4      h = head; t = tail;
5      next = h.next;
6      if (h != head) continue;
7      if (h == t) {
8        if (next == null) // Is queue empty?
9          return null;
10       // Try to advance tail.
11       CAS(&tail, t, next);
12     } else {
13       // Try to advance head.
14       if (CAS(&head, h, next))
15         return next.value;
16     }
17   }
18 }
```

Figure 3.4: Pseudo-code for removing a node from the queue.

3.1.2 The Dequeue Operation

The dequeue operation is depicted in Figure 3.4, and it 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.

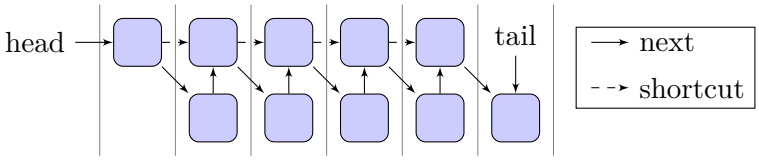

Figure 3.5: Idealized trace of a queue with shortcut distance $n = 2$. Each column is traced in a single cycle. Such a queue can utilize a maximum of two processors.

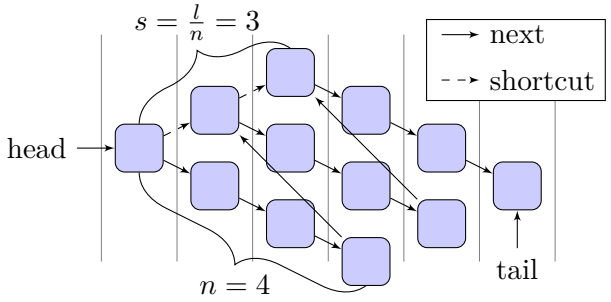

Figure 3.6: Idealized trace of a queue with $l = 12$ nodes, and shortcut distance $n = 4$. Each column is traced in a single cycle.

3.2 Setting the Shortcut Distance n

An important question for the design of the shortcuts is what distance should be set between the nodes in the shortcut list. The optimal distance between shortcuts depends on the length of the list, and on the number of processors. Choosing a distance too small (as in Figure 3.5, for example) allows parallelization only for a small number of processors. The shortcut distance can also become too large, again limiting the parallelism. In this section we calculate an optimal distance.

Suppose we optimize for the idealized trace measure described in section 1.1. Namely, we ignore load balancing issues and just want to reveal nodes early on in the trace so that all processors have enough work. The actual distance between nodes of the shortcut list varies due to concurrent execution as explained in section 3.3. But for the sake of simplicity let us assume here that all distances are exactly n. Consider a list of length l and assume for simplicity that the number of processors is very large, and that that l is divisible by n. Now, let us compute the number of (parallel) steps it takes to scan a list with l nodes and with shortcut length n.

Consider an (imaginary) partition of the list to sublists of size n, having a sublist starting at each shortcut in the shortcut list. There will be $s = l/n$

such sublists. See an illustration in Figure 3.6. The maximum number of objects that the trace will be able to scan in parallel, in a single cycle, is $\min\{n, s\}$. A parallel trace of such a list would start with its head and open up more and more parallelism as more nodes are discovered via the shortcuts. Assuming a large number of tracing cores, we see that after $\min\{n, s\}$ cycles in which the number of parallel objects scanned increases per cycle, a steady state is reached in which $\min\{n, s\}$ objects are traced in parallel in each cycle. Now, the steady state holds for $\max\{n, s\} - \min\{n, s\}$ cycles until the list is almost completely traced, and then begins a final phase of $\min\{n, s\} - 1$ cycles in which the number of parallel objects scanned reduces in each cycle. Overall, we get that the number of cycles required for the scan with a large number of cores is

$$\max\{n, s\} + \min\{n, s\} - 1 = n + s - 1 = n + \frac{l}{n} - 1$$

An optimal shortcut distance would therefore be $n = \sqrt{l}$. However, this assumes that the number of processors is larger than $\sqrt{l}$. If it is not, then the number of cycles will grow accordingly, and one can set n to the number of processors without affecting the idealized trace measure. For testing garbage collectors in practice, we chose to set n to (approximately) $\sqrt{l}$ even with a smaller number of processors, since the load balancing isn't optimal, and having more parallelism available to the garbage collector could be helpful.

3.3 Correctness and Analysis

We first claim that the proposed enhanced queue (with the shortcuts) is lock-free. Recall that the original queue is lock-free. None of our modifications change any of the original data structure fields, so we only need to show progress when threads are running the additional code, i.e. the `addShortcut` method.

We name the list of nodes that have a non-`null` `shortcut` field the *shortcut list*. We need to show that adding a node to the shortcut list is a lock-free operation. Since our `addShortcut` method is very similar to the original enqueue operation, our proof is similar to the proof in [26]. The proof depends on the following invariant.

Invariant 1 *The `shortcut_tail` field always points to a node that is either the last in the shortcut list, or the second to last.*

According to this invariant, either `shortcut_tail` points to a node

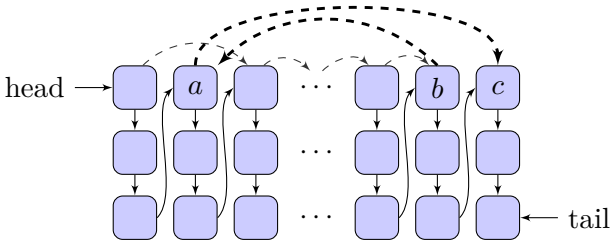

Figure 3.7: Backwards shortcut edges scenario illustration. The solid edges represent `next` references, while the dashed edges represent shortcuts. The shortcut between the nodes b and a goes backwards, and the shortcut between the nodes a and c can skip an unbounded number of list nodes.

whose `shortcut` field is `null`, which is the last node in the shortcut list, or it points to a node whose `shortcut` field points to that last node.

Proof. This invariant is initially true, and it is kept by the enqueue operation. It could have been broken either when changing the `shortcut` field of a node, at line 8 in the `addShortcut` method (Figure 3.3), or when updating the `shortcut_tail` pointer at lines 11 or 14. The changes to the `shortcut` field only occur to a node that is pointed by `shortcut_tail` and has a `null` `shortcut` field, meaning that after this change `shortcut_tail` must point to the second-to-last node in the list. When changing `shortcut_tail`, it is always to the next node in the list, therefore moving from the second-to-last node to the last one. ■

We now prove liveness, using the invariant. A thread can take another iteration in the `for` loop (line 2) in the `addShortcut` method, either at line 5, or by failing the checks at line 6 or line 8. If a thread executes the `continue` statement at line 5 more than once, other threads must have changed the `shortcut_tail` reference twice. Because of the invariant, this means that another node must have been inserted during this period.

If a thread fails the check at line 6, `shortcut_tail` must point at the second-to-last node. After our thread performs line 11, the `shortcut_tail` must point to the last node (regardless of whether the CAS succeeded). If our thread fails the check at line 6, another thread must have added another node. Finally, if the CAS operation at line 8 fails, then another thread must have changed the `shortcut` field and added a new node.

Having shown lock-freedom, we now analyze the shortcuts' distances. Although the data structure attempts to maintain a fixed distance n between shortcuts, the actual distance can vary due to the effects of concurrent interaction. The shortcut field might point to a node whose distance is larger

or smaller than n. In fact, there are concurrent scenarios that can make a shortcut edge go backwards in the linked-list. Let us explain such scenarios.

A thread that attempts to insert a new node to the shortcut queue might be delayed after inserting the node to the queue but before adding it to the shortcut list. After any unbounded number of insertions, the thread might wake up and insert the node (which is now very much behind the current `tail`) to the shortcut list. This means that the new `shortcut` will go backwards in the list, and furthermore that the backwards distance is not bounded. The next node that is inserted to the shortcut list will follow the one inserted by the delayed thread, it will point forwards again, but the distance between these two cannot be bounded as well. This scenario is illustrated in Figure 3.7.

Chapter 4

Linked-Lists

While we decided to focus on the concurrent queue data structure for this paper, we nevertheless wanted to show how GC scalability problems in other kinds of linked-lists can be solved. We look at two such cases: linear linked-lists, and immutable linked-lists.

4.1 Linear Linked-Lists

In linear linked-lists, elements can be accessed, removed, or inserted by their ordinal position in the list. We examined two alternatives for linked lists used with linear operations, that could provide more parallelism for the garbage collector. First, we implemented a skip-list with linear list operations [28] as an alternative to the `java.util.LinkedList` class for use with the `bloat` benchmark. The required access to the list in this benchmark is a serial access according to the ordinal number of the element. Assuming that the application remains sequential, we can still help the garbage collection scale by using a skip-list instead of a linked-list. To this end, we let each index node in the skip-list contain the number of elements in the sub-list that is between its node and the next index's nodes. This allows lookup, insertion and deletion in $O(\log l)$ complexity (on average), where l is the length of the list, and the skip-list references allow the garbage collector to scale well.

Another alternatives to the `java.util.LinkedList` class we tested was a list of arrays. It is a linked-list, in which each node contains a fixed-size array of list elements. When inserting or deleting an element, we first iterate through the list to find the node to which the change belongs. We then modify the array in that node, moving elements in the array as necessary. If we need to insert to an array that is already full, we split the array into to equal halves. If we remove a node from an array leaving it empty, we delete

its node from the list. The complexity of lookup in this data structure is $O\left(\frac{l}{n}\right)$ where n is the size of the arrays. For insertions and deletions, the complexity is $O\left(\frac{l}{n}+n\right)$ The slightly modified skip-list and the list of arrays were implemented with `bloat` and we report the results in chapter 5.

4.2 Immutable Linked-Lists

A different problematic linked list that we found was used by the `javac` benchmark from the SPECjvm2008 benchmark suite. In that benchmark there were immutable linked-lists, which are also common in functional programming languages. The benchmark uses standard list operations such as `nil()`, which returns the empty list node; `cons(head, tail)`, which constructs a list with the given head as the first element, followed by the list given as tail; `head(list)`, which returns the head item in the list; etc.

Immutable long linked-lists cause scalability problems for the garbage collection exactly like mutable linked-lists do. However, solving this problem for immutable linked-lists may be easier. Note that immutable linked-lists do not pose a problem when defining their semantic for concurrent operations. Therefore, our reservations for handling concurrent non-sorted linked-lists do not apply. Since this only happened for one of our benchmarks, and in this benchmark its use is ad-hoc and difficult to replace with a library function, we did not implement a solution for it, but in what follows we would like to propose a modification that adds shortcuts to the list, and might alleviate the problem.

The solution is to add a `shortcut` field to each node in the list. Also, in addition to the head pointer of the list, we extend the header by adding a pointer denoted `shortcut_head` to the node that has the most recently added shortcut, and we also add a counter to the header, which counts the number of nodes between the head node and the node pointed by the `shortcut` pointer. In a `cons` operation, when adding a new node at the beginning of the list, the list `counter` value is checked to see whether it equals the desired shortcut distance n. If so, the new node's `shortcut` field is set to the `shortcut_head` pointer to create the shortcut, the `counter` is set to zero and the `shortcut_head` is set to point to the new node. Otherwise, we only increment `counter`. The reason this is not easy to implement with `javac` is that the benchmark considers the header as a regular node, while we need to extend it to include the counter and the additional pointer. Thus, extensive modification of `javac` is required to make it work with such a modified implementation, and this structure would not work with any of the other benchmarks we investigated.

Chapter 5

Measurements

In order to check the benefits of the proposed new queue, we ran the benchmarks presented in chapter 2, and computed the idealized trace utilization for the original benchmarks and for benchmarks modified to use the queue with shortcut references as proposed in chapter 3. We present these results in section 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 section 5.2. Finally, in section 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, as discussed in section 3.2. 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 li-

brary's `ConcurrentLinkedQueue`, written by Doug Lea. This implementation is different from the Michael and Scott algorithm in that it uses garbage collection, so it doesn't 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.

As mentioned in chapter 3, we also implemented a variant of the shortcut queue that added shortcuts in a probabilistic manner to each node with probability $1/n$. This version consistently performed worse than the deterministic version and we omit these results here for lack of interest. The full set of results can be found in chapter 7.

5.1 Idealized Trace Utilization

In Figure 5.1 the idealized trace utilization of the benchmarks described in chapter 2 is compared with executions that make use of shortcuts. For the `xalan`, `Jetty` and `mtrt` we used the shortcut queue described in chapter 3. 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 chapter 3. Figure 5.1'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 wasn't dramatic, there was always a drastic improvement with the lower number of processors. The `pmd` benchmark also improved its utilization, but the more significant

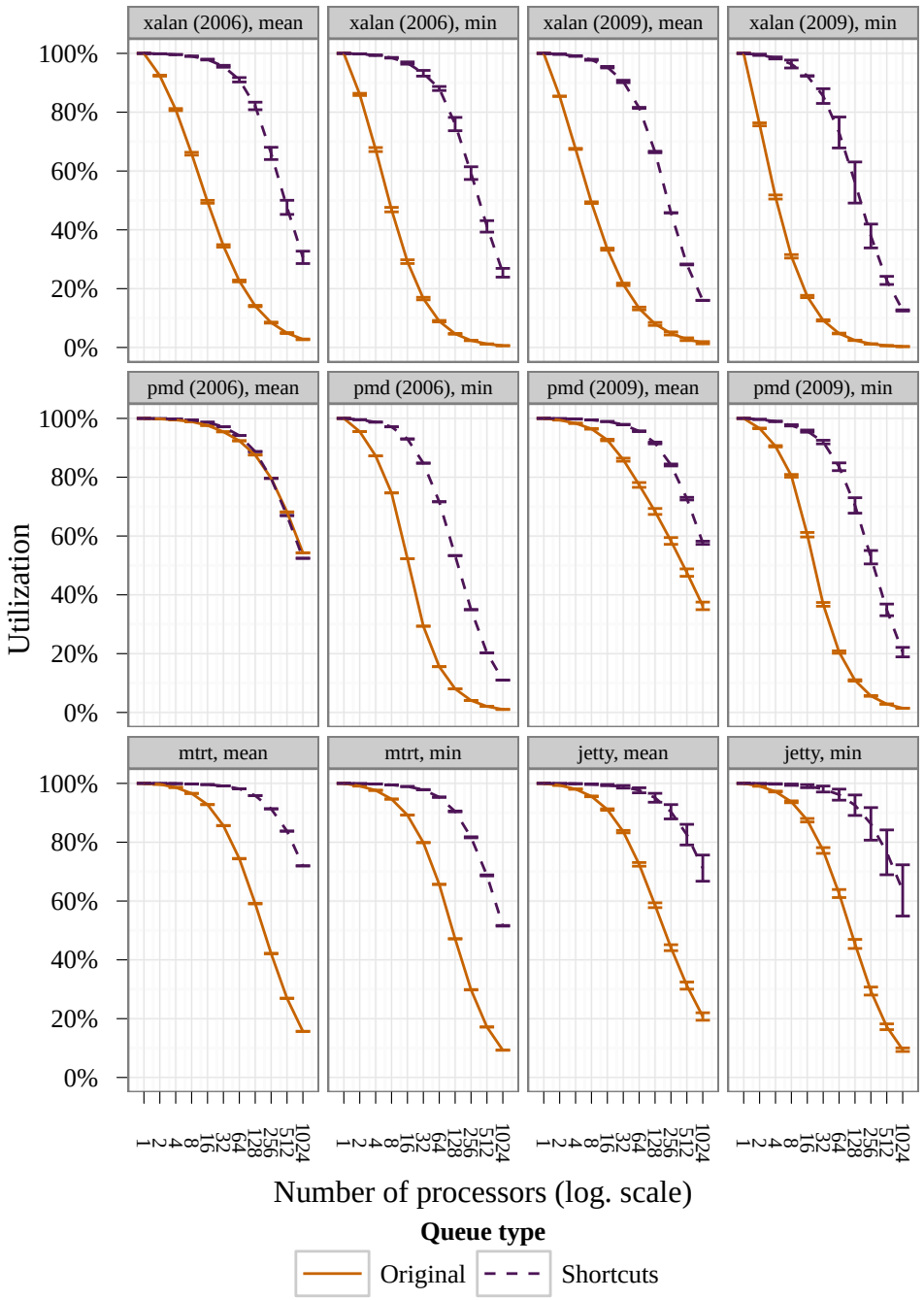

Figure 5.1: Idealized trace utilization for the original and modified benchmarks

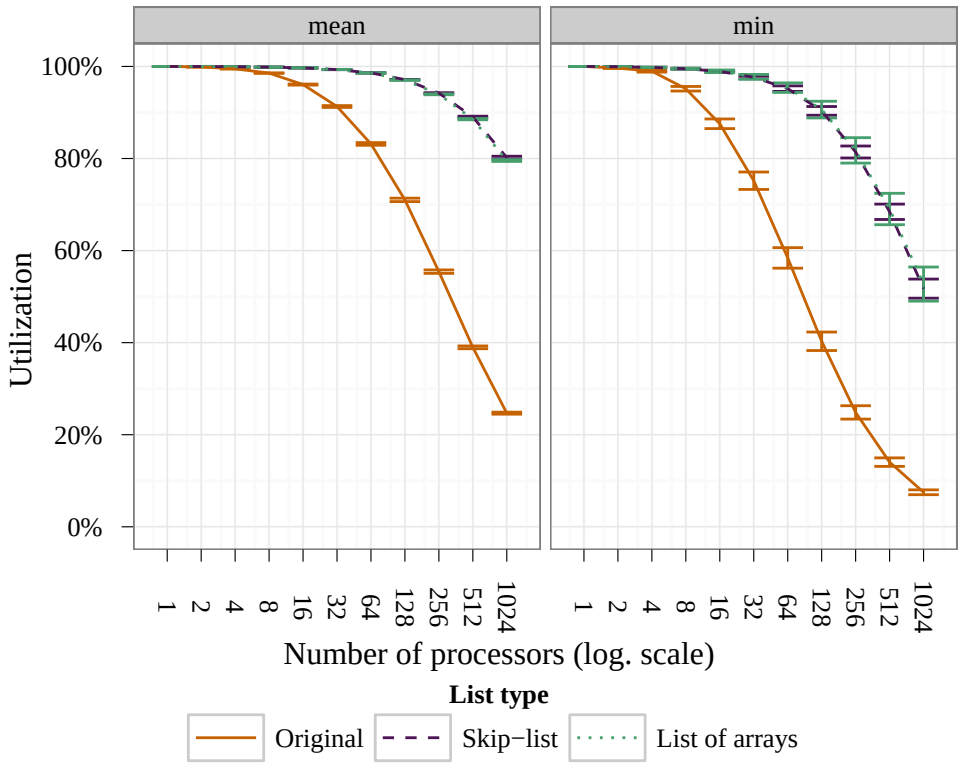

Figure 5.2: Idealized trace utilization for the original and modified `bloat` benchmark.

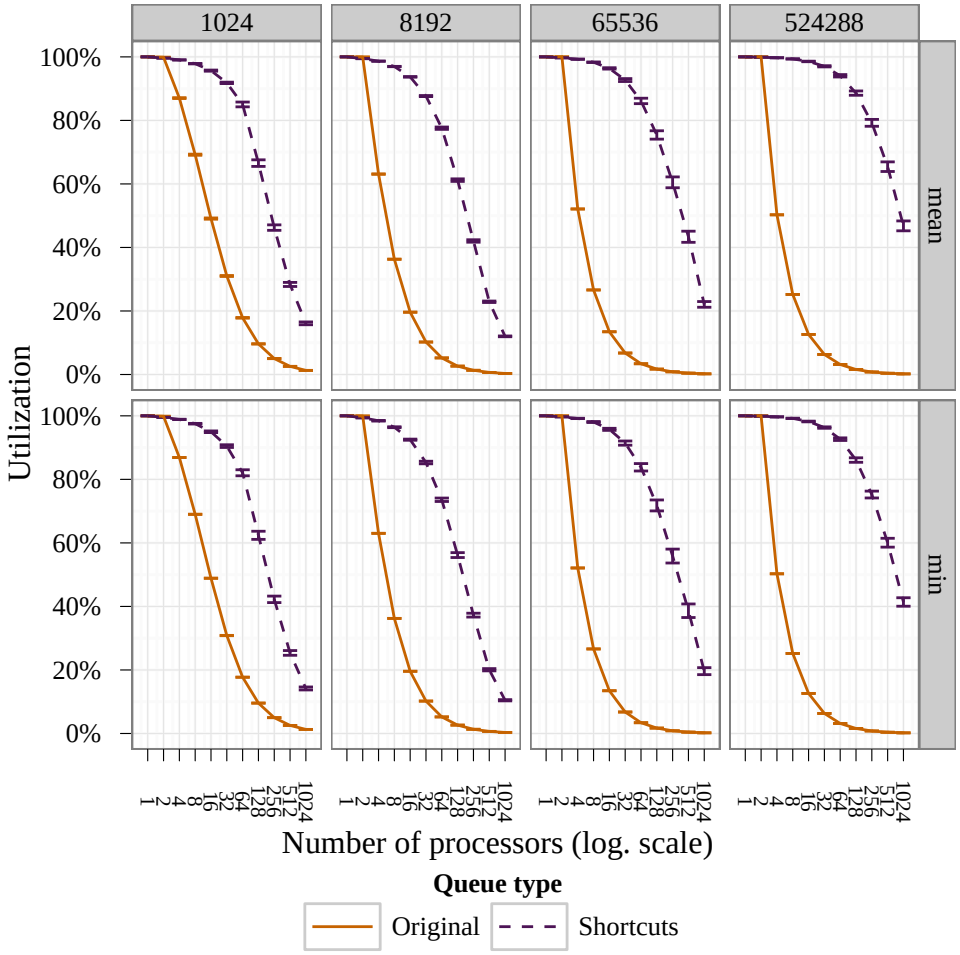

Figure 5.3: Idealized trace utilization for the artificial benchmark

improvement was with the minimal value, while the mean utilization didn't 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.

Next, we looked at the `bloat` benchmark, which employs a generic linked-list. We attempted to improve it by replacing the linked-list with the skip-list and the list of arrays as described in section 4.1. In Figure 5.2 we present the idealized trace utilization measure of the `bloat` benchmark (which employs `java.util.LinkedList`) against a version of bloat that employs a skip-list, and a version with a list of arrays. The modified versions achieved a drastic improvement in the idealized tracing utilization measure.

Finally, we ran our artificial benchmark with different queue lengths, see

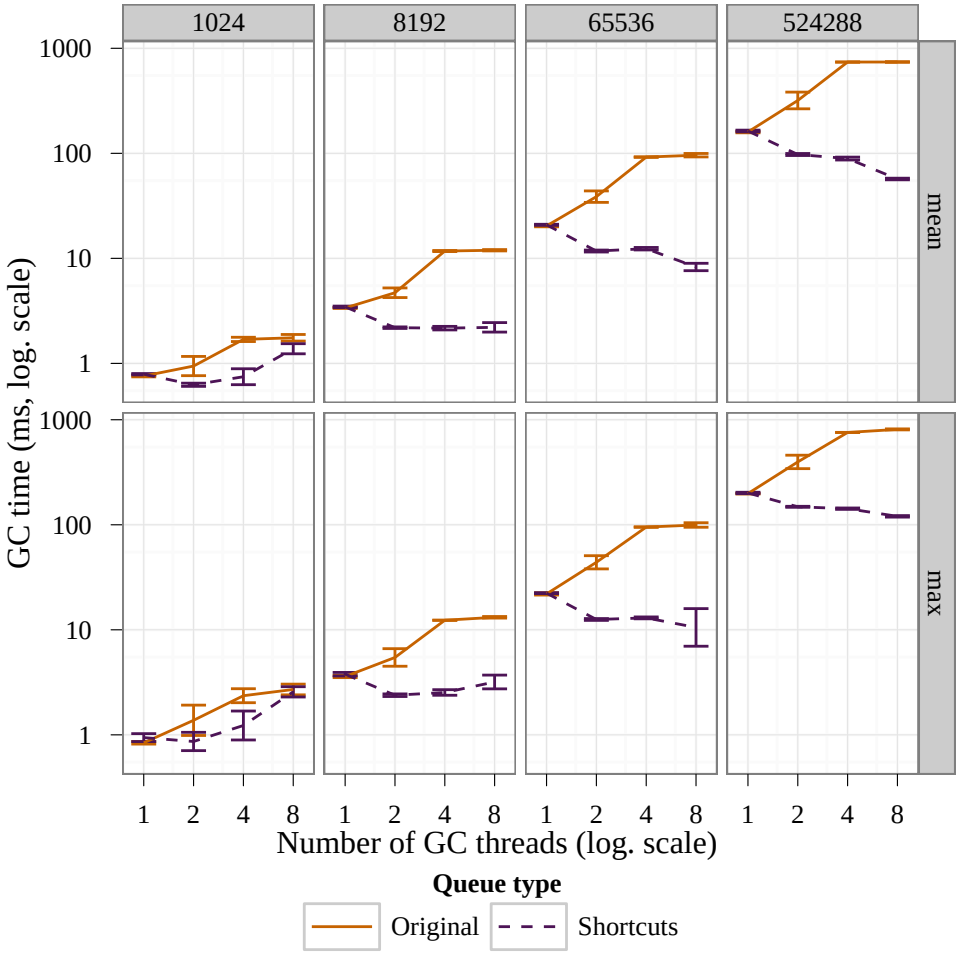

Figure 5.4: Artificial benchmark garbage collection time under the HotSpot JVM

Figure 5.3. Each column in the grid stands for a different queue length. The first row shows results of the mean idealized trace utilization measure, while the second row depicts results with the minimum utilization. The figure shows how the scalability problem becomes more acute as the length of the queue grows, and how using shortcuts alleviates the problem.

5.2 Garbage Collection Time

Figures 5.4 and 5.5 show the garbage collection time for executions of the artificial benchmark. Since this measurement is very sensitive to the garbage collection implementation (which we do not control), we checked the times both for the HotSpot JVM and for IBM's JVM. The chart shows in each

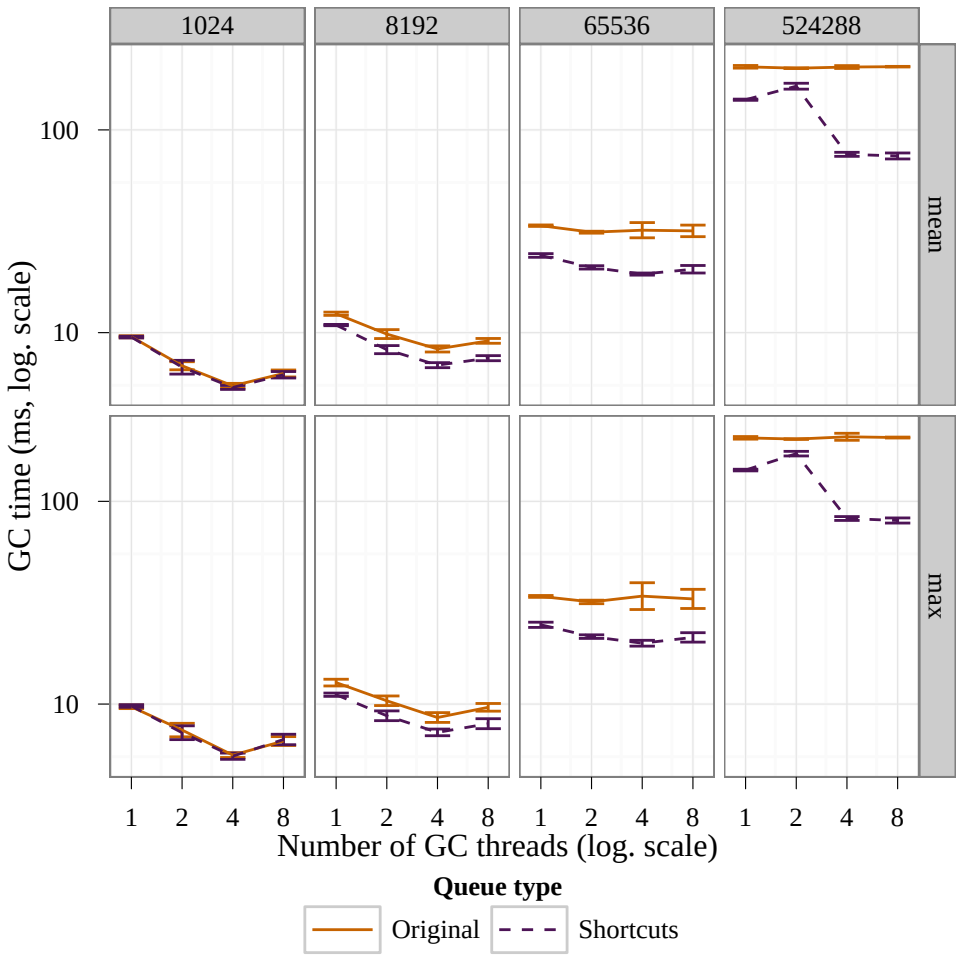

Figure 5.5: Artificial benchmark garbage collection time under IBM's JVM

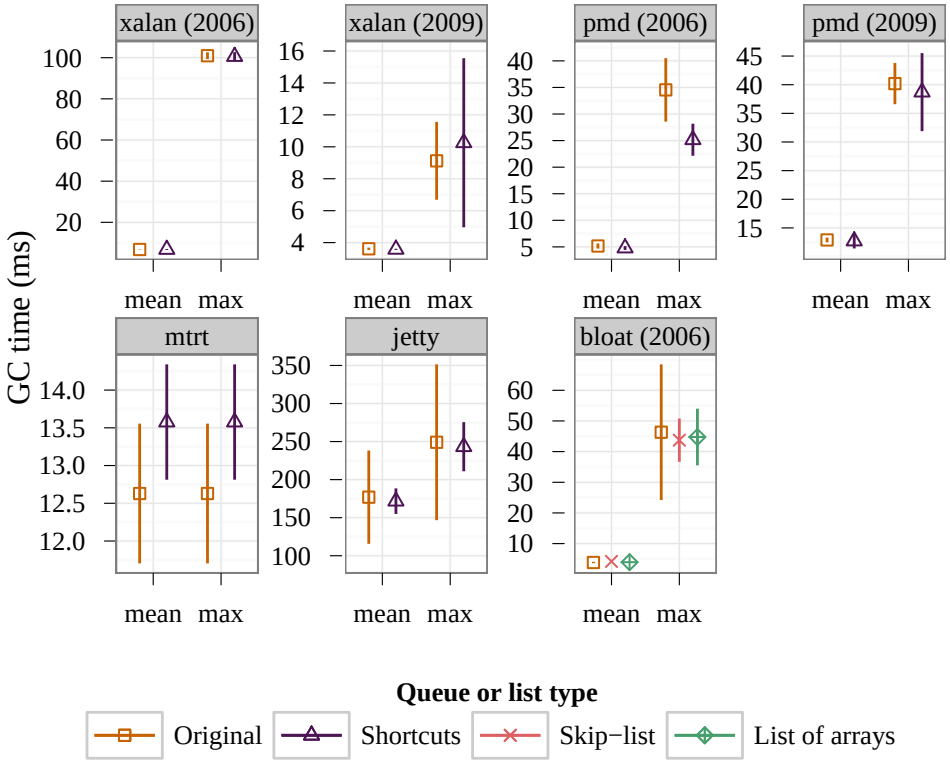

Figure 5.6: GC pause time for the original and modified benchmarks under the HotSpot JVM.

column a different target queue length, with the first row showing the mean GC time, and the second row showing the maximum GC time. Each cell in the chart shows the 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 [20], 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 [23], and doesn't 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.

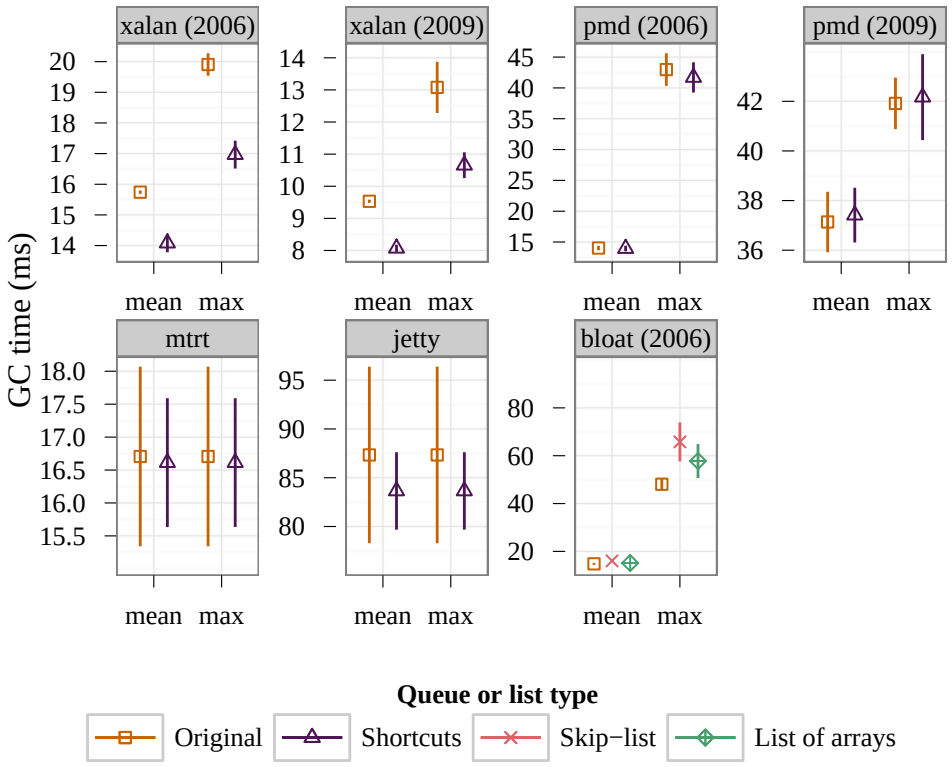

Figure 5.7: GC pause time for the original and modified benchmarks under IBM's JVM.

We also compared the GC pause times for the other benchmarks using the enhanced data structures, under HotSpot and IBM's JVMs (Figures 5.6, and 5.7). 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.

On a different front, neither the use of skip-lists nor the list of arrays for the `bloat` benchmark were a success. Under IBM's JVM, `bloat` suffered from about 40% increase in the maximum GC time, when using a skip-list instead of an ordinary linked-list, and a 30% increase when using the list of arrays data structure. To explain this loss, we note that the `bloat` benchmark doesn't reach a very deep heap shape. The coarser grained load balancing mechanism in IBM's JVM doesn't allow the GC to benefit from the extra references, and in this case the shortcuts and the extra index objects used by the skip-list only cause extra work for the tracing threads.

The `bloat` benchmark also suffered a degradation in throughput, as can be seen in section 5.3. This highlights the difference between the cost of switching to a skip-list and the cost of switching to our specially designed lock-free queue with shortcuts. Although skip-lists could be used to replace queues as well, the performance cost for doing that would be much higher.

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 (Figures 5.8, and 5.9). 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.

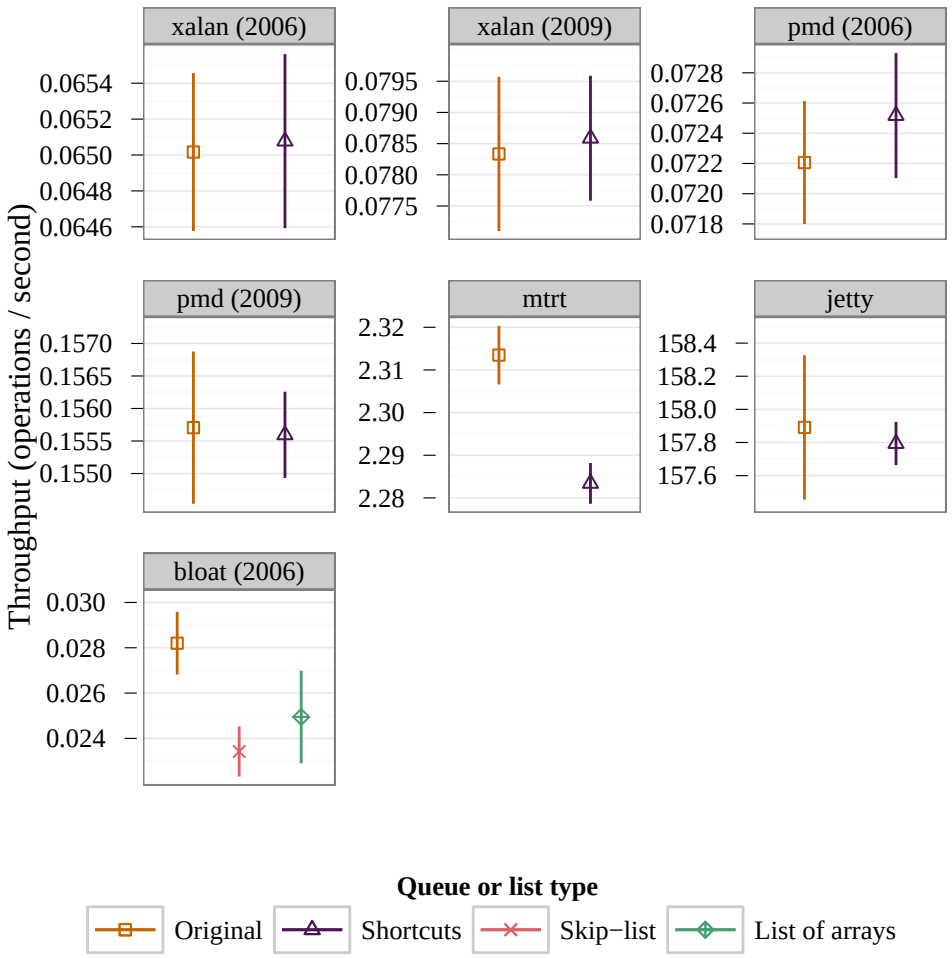

Figure 5.8: Throughput of the original and modified benchmarks under HotSpot JVM.

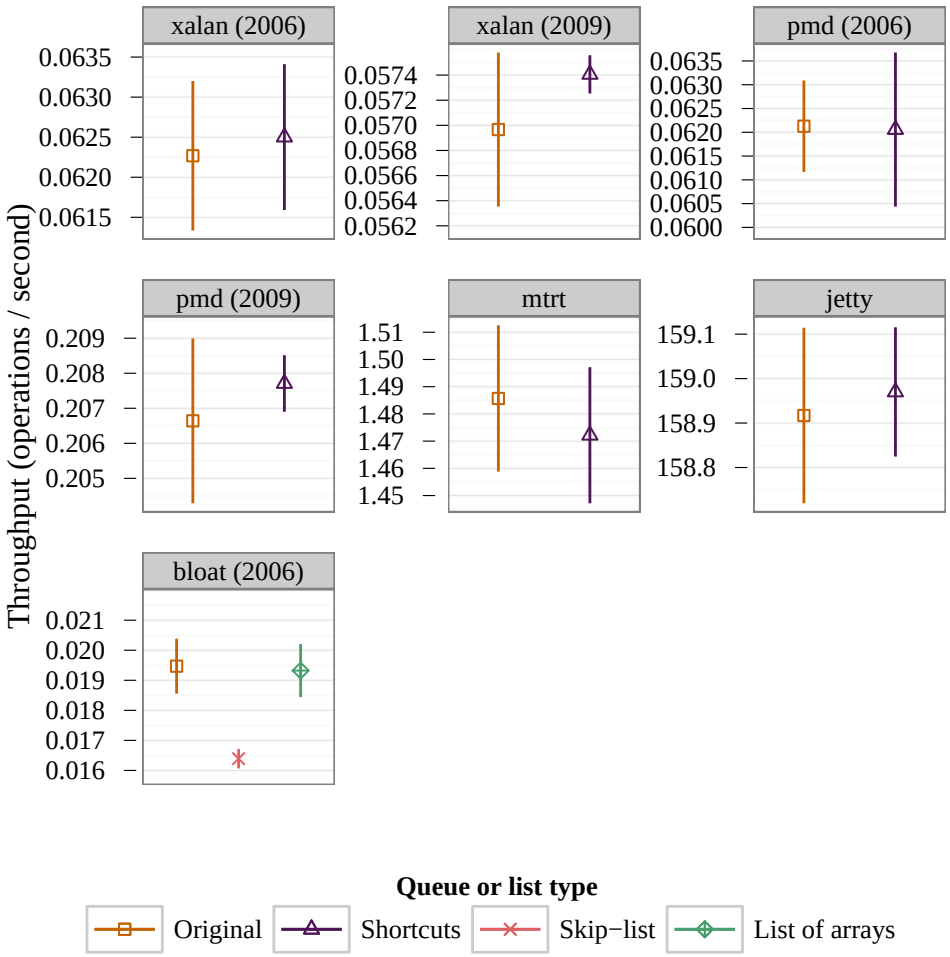

Figure 5.9: Throughput of the original and modified benchmarks under IBM's JVM.

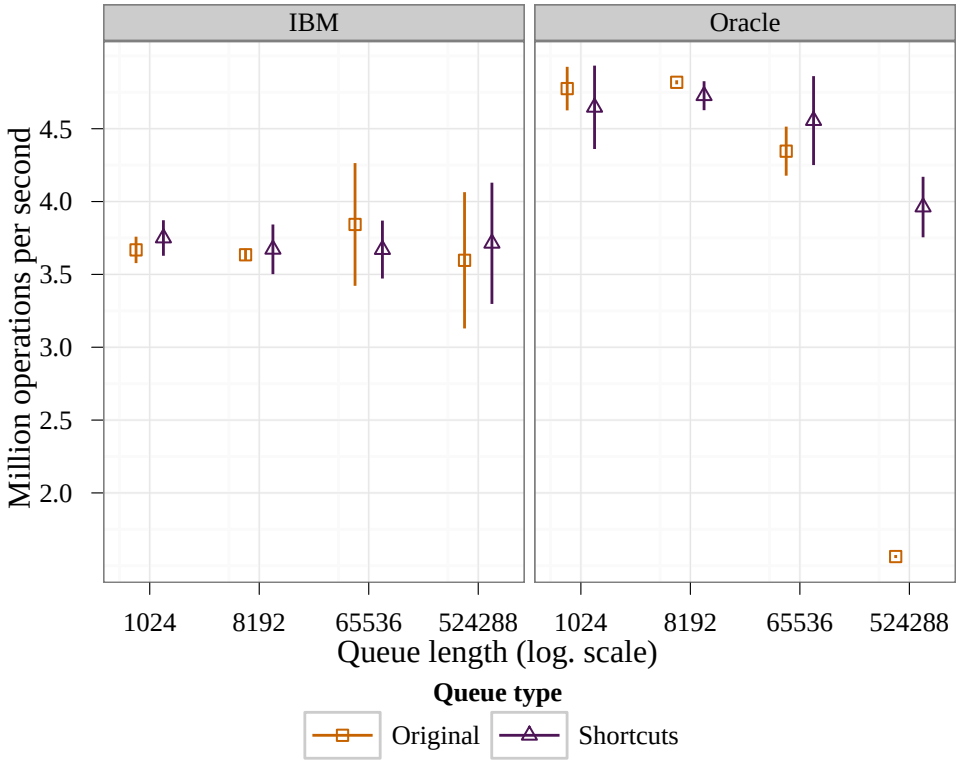

Figure 5.10: Artificial benchmark throughput (queue operations per second) by JVM, queue length, and queue type.

We also compared the performance of the artificial queue benchmark with and without the shortcuts on our eight-core machine (Figure 5.10). 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 section 5.2.

Chapter 6

JVM Owned Linked-Lists

Surprisingly, when we started measuring the benchmarks, we discovered more problematic heap shapes than the heap shapes found in previous work [31, 5]. It turned out that some long linked-lists are actually generated by the JVMs and not by the benchmarks. In particular, long linked-lists are created because the finalizers were implemented using a linked-list that held all objects with finalizers. This showed up in Oracle HotSpot JVM and not in the Jikes JVM, which uses a dynamic array for that purpose. In addition, the HotSpot class loader keeps a linked-list of directories available in each `jar` file, which turned out long for the DaCapo 2006 benchmark suite (see Figure 6.1). In the main body of the paper we reported the idealized trace utilization measure for the heap excluding these lists, as we did not consider them part of the benchmarks. In this chapter, we report measures that include the additional lists. As can be seen some of the heap shapes are much worse. See for example, the maximum depth of the `javac` and the `compress` benchmarks in SPECjvm98, which had a large number of objects in their finalizer list, or the benchmarks of the DaCapo 2006 suite, which all had a depth of at least 269 objects, because of the list in their class loader data.

Benchmark	Mean depth	Max depth	Heap-dumps
pmd	565.0	28599.5	147.5
xalan	4241.3	8399.4	42.9
eclipse	303.4	1230.6	34.4
bloat	368.8	733.6	49.5
lusearch	289.3	335.0	23.2
antlr	269.0	269.0	29.2
chart	269.0	269.0	7.5
fop	269.0	269.0	27.8
hsqldb	269.0	269.0	11.5
jython	269.0	269.0	23.2
luindex	269.0	269.0	12.5

Table 6.1: Maximum depth of an object in DaCapo 2006

Benchmark	Mean depth	Max depth	Heap-dumps
pmd	4104.8	33587.6	17.3
xalan	4257.7	8432.3	66.1
avrora	985.1	1907.8	19.8
tradesoap	1031.2	1063.5	12.8
tradebeans	1034.6	1041.4	6.8
fop	466.0	868.5	10.5
eclipse	530.0	836.6	5.0
batik	250.2	311.0	77.8
tomcat	171.2	272.9	9.2
jython	86.0	86.0	9.8
lusearch	55.1	73.2	139.1
h2	64.9	70.5	12.7
luindex	41.8	45.9	6.5
sunflow	42.3	45.7	14.2

Table 6.2: Maximum depth of an object in DaCapo 9.12

Benchmark	Mean depth	Max depth	Heap-dumps
javac	8591.7	8807.7	24.1
mtrt	1406.2	1413.0	11.3
raytrace	1405.0	1413.0	6.1
compress	477.4	490.0	133.5
jack	305.2	309.1	5.5
db	143.0	143.0	11.2
mpegaudio	95.7	97.6	12.0
jess	78.5	79.8	5.3

Table 6.3: Maximum depth of an object in SPECjvm98

Benchmark	Mean depth	Max depth	Heap-dumps
xml.validation	1180.9	1184.0	39.4
compiler.sunflow	563.0	669.0	53.1
compiler.compiler	429.7	519.9	30.7
crypto.rsa	424.7	512.9	92.6
crypto.signverify	390.9	442.2	223.3
xml.transform	260.1	281.3	96.7
monte_carlo	212.2	240.1	14.2
serial	146.8	171.5	45.8
derby	83.0	133.4	18.8
crypto.aes	64.8	75.5	202.7
scimark.fft	44.4	65.1	15.9
sunflow	49.1	65.1	122.2
scimark.sor	46.1	62.5	13.9
scimark.lu	44.4	58.9	11.3
mpegaudio	46.6	50.0	8.0
compress	43.0	43.0	13.0
scimark.sparse	38.1	38.3	13.1

Table 6.4: Maximum depth of an object in SPECjvm2008

Benchmark	Mean depth	Max depth	Heap-dumps
jetty	2669.6	3495.0	33.8

Table 6.5: Maximum depth of an object in `Jetty`

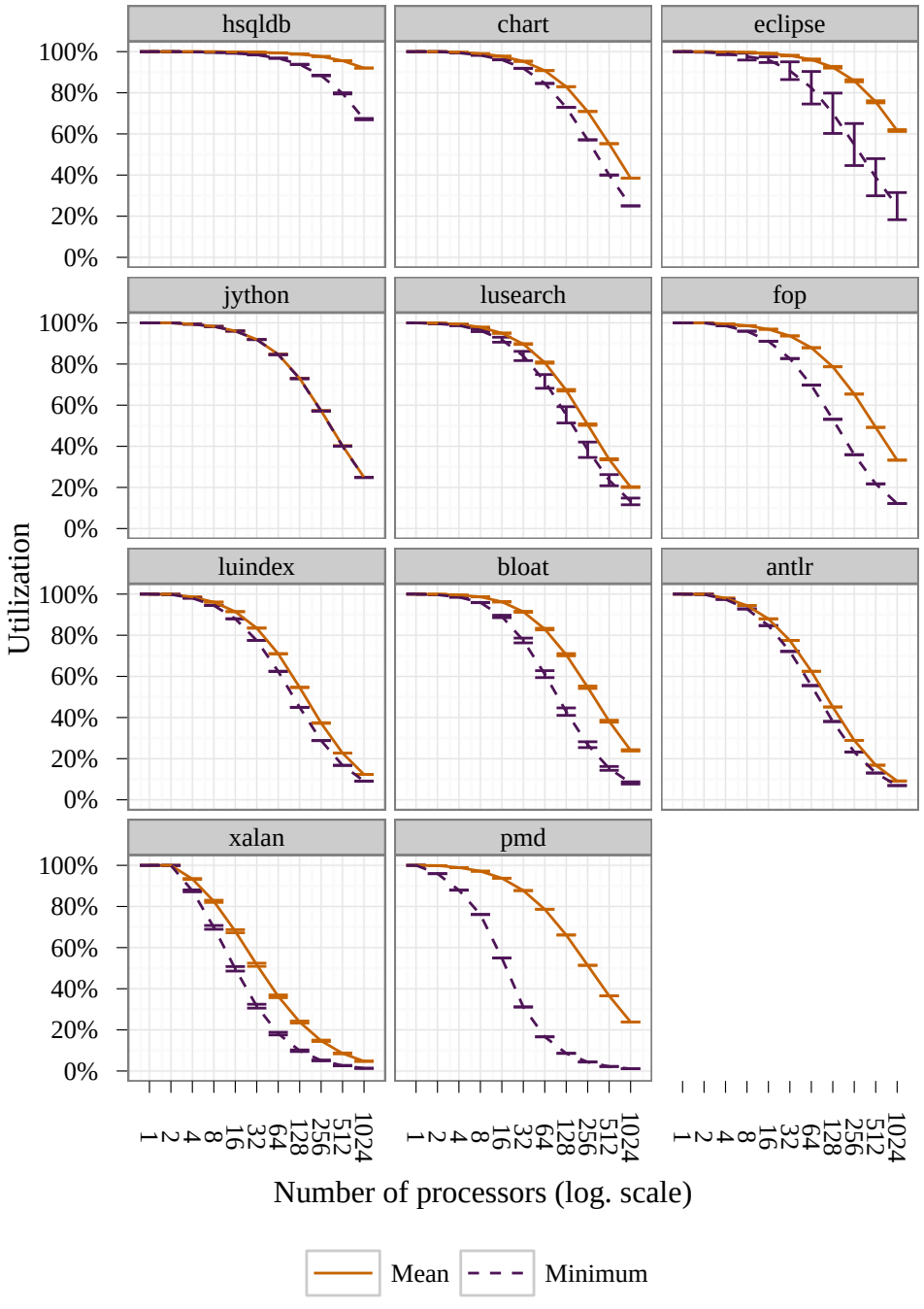

Figure 6.1: Idealized trace utilization, DaCapo 2006 benchmarks

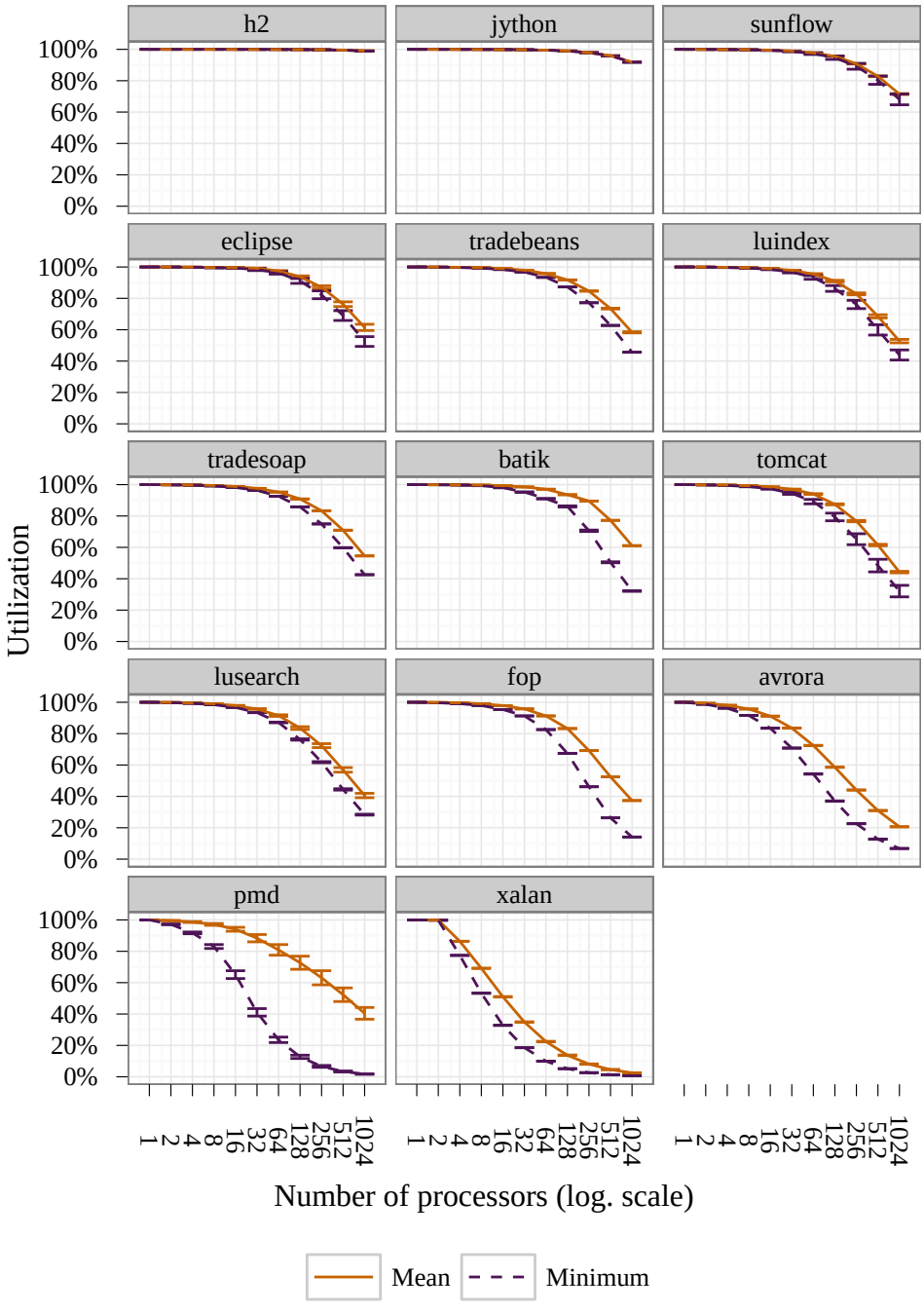

Figure 6.2: Idealized trace utilization, DaCapo 9.12 benchmarks

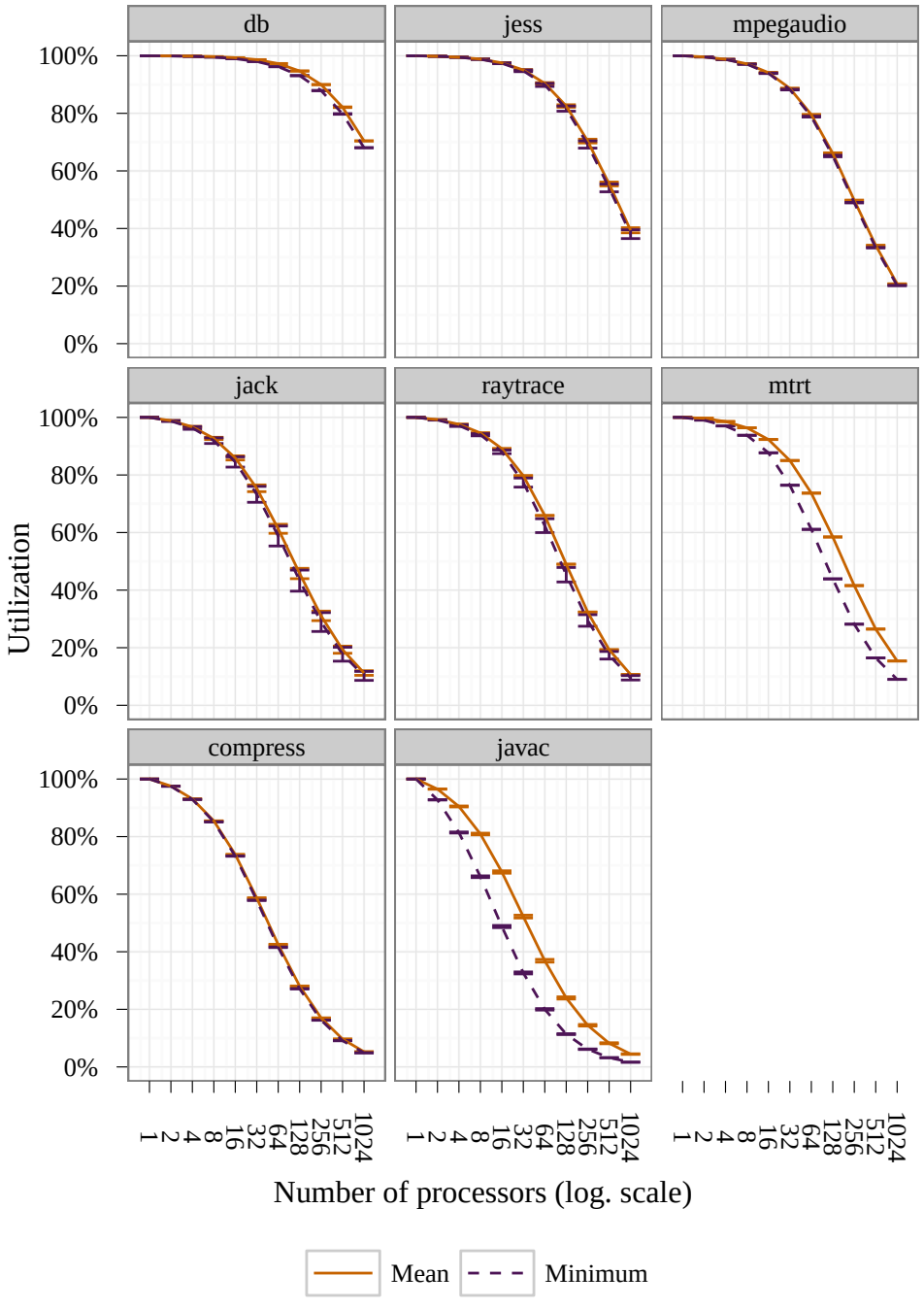

Figure 6.3: Idealized trace utilization, SPECjvm98 benchmarks

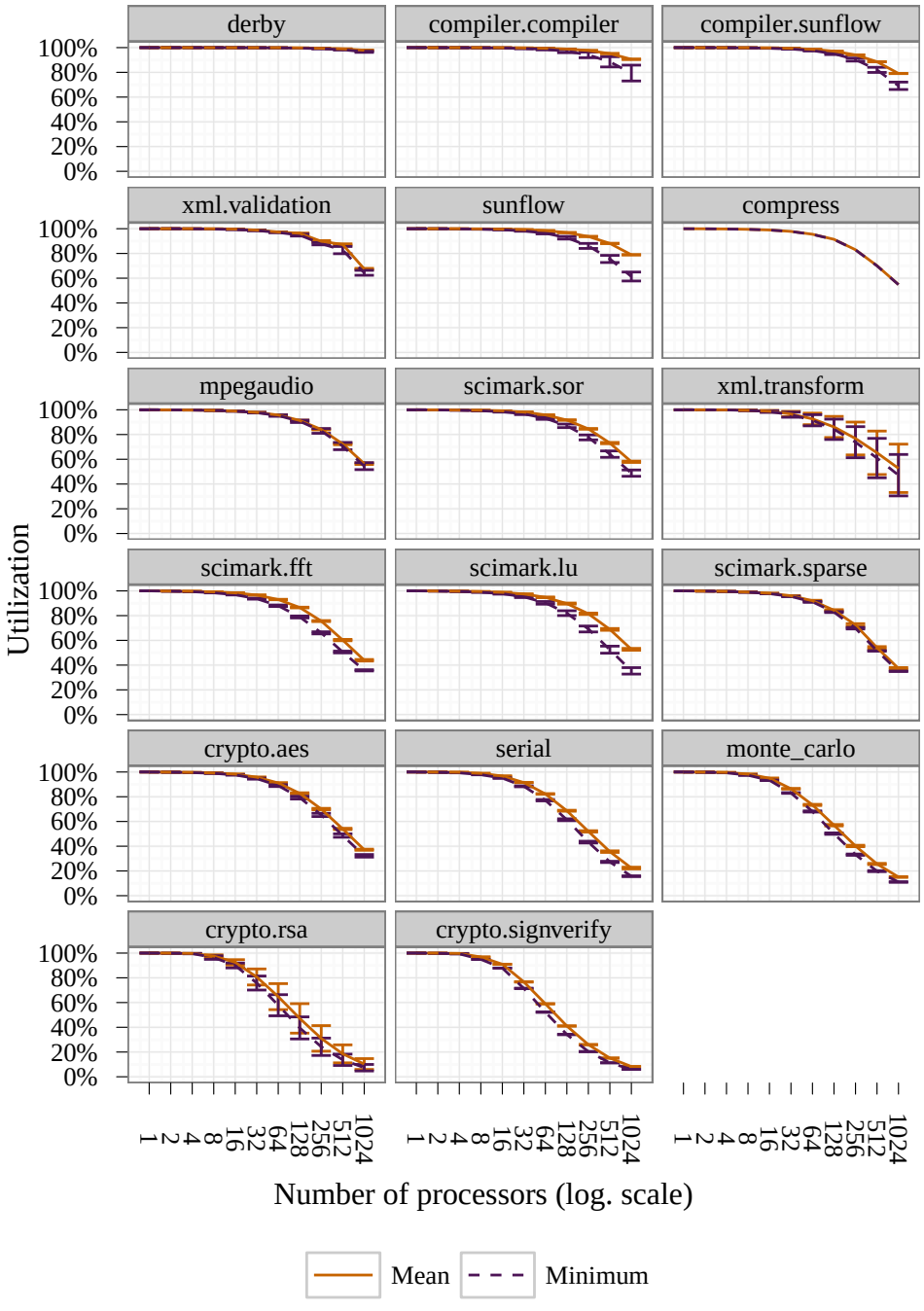

Figure 6.4: Idealized trace utilization, SPECjvm2008 benchmarks

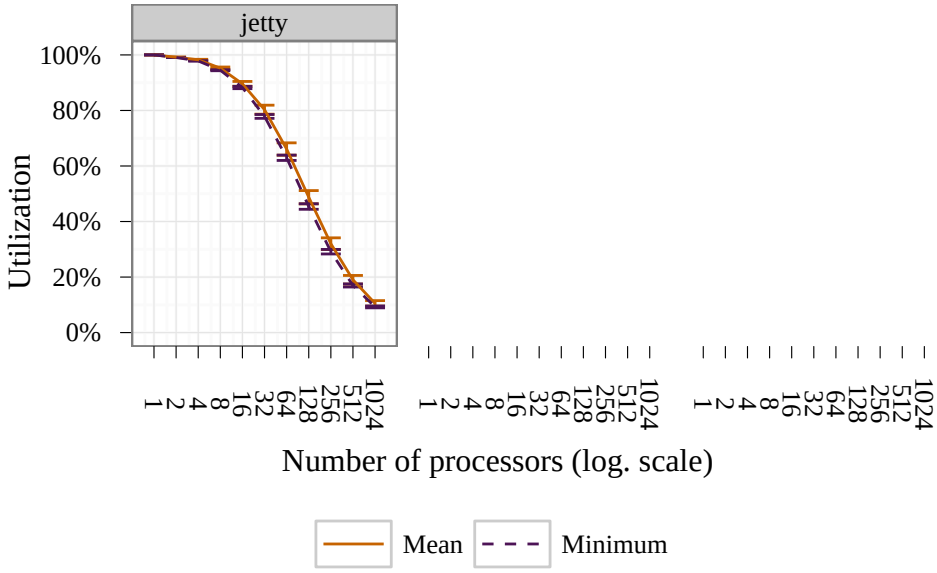

Figure 6.5: Idealized trace utilization, `Jetty` benchmarks

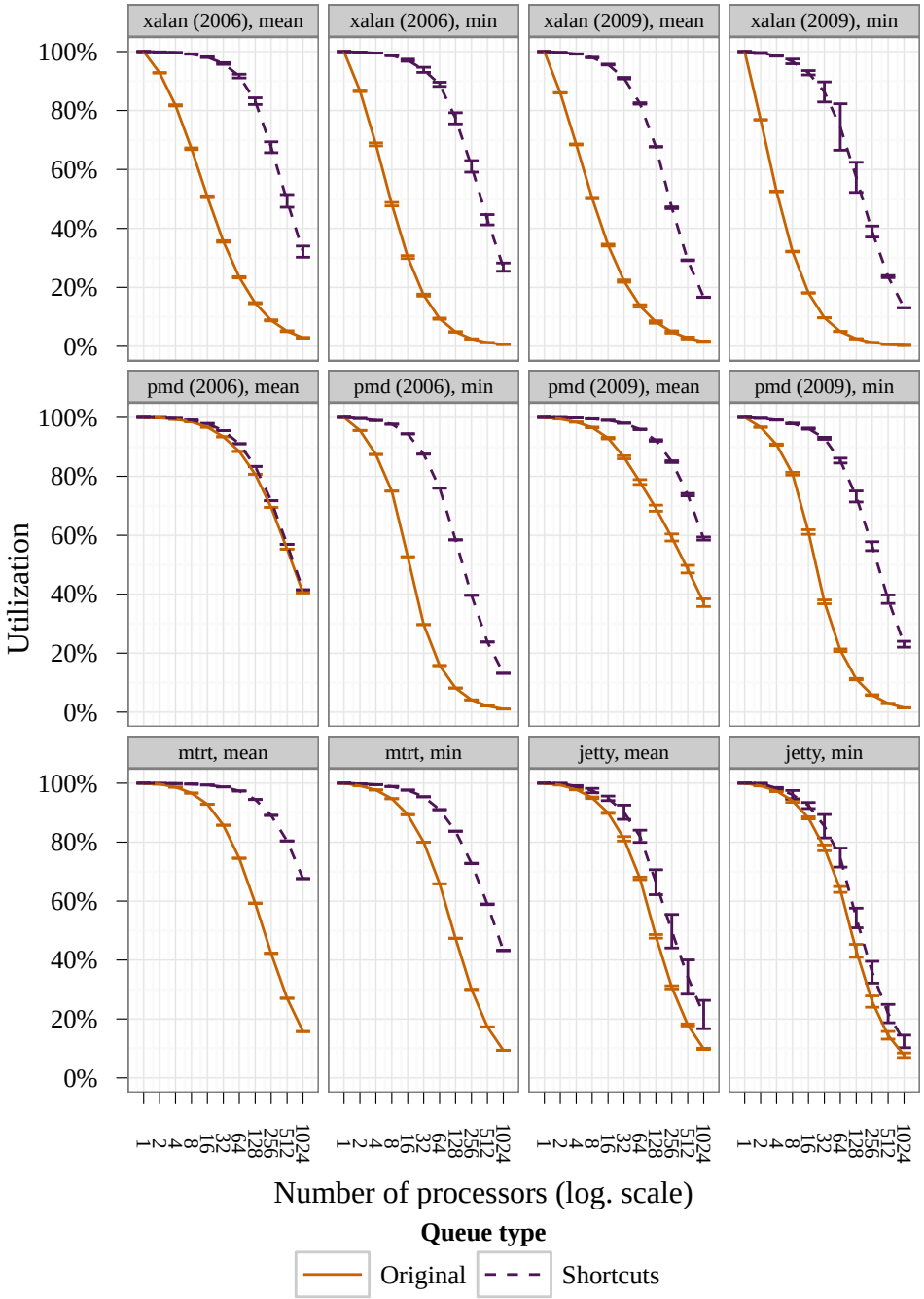

Figure 6.6: Idealized trace utilization for the original and modified benchmarks

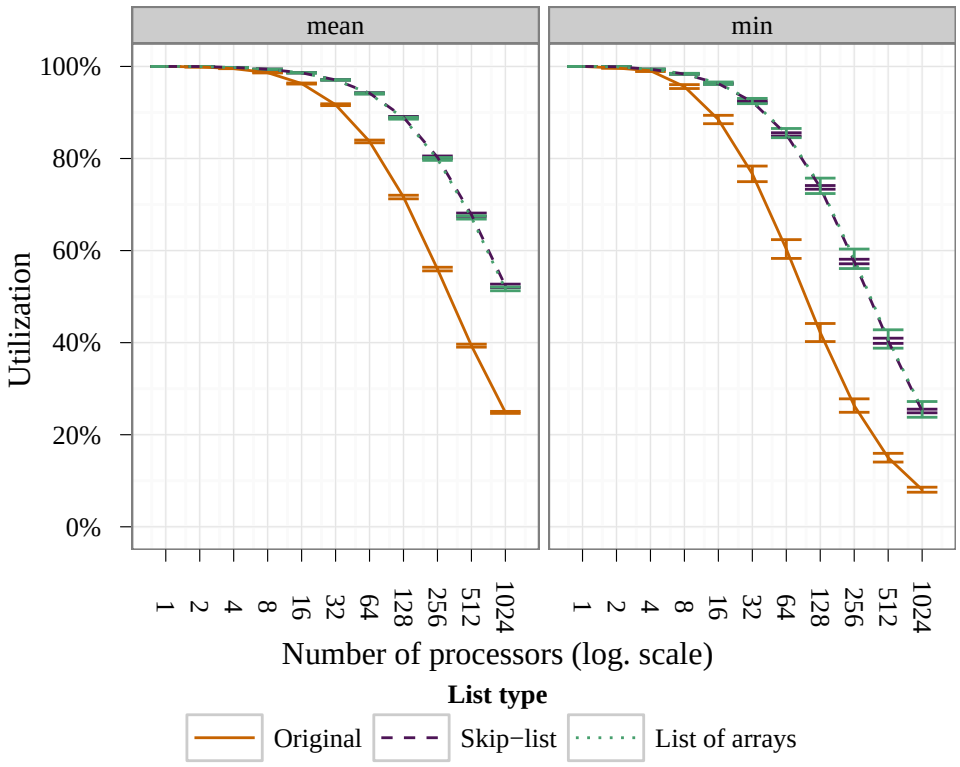

Figure 6.7: Idealized trace utilization for the original and modified `bloat` benchmark

Chapter 7

Randomized Shortcut Insertion Measurements

We have created a variant of our queue with shortcuts, that uses a random number generator to add a shortcut with probability $p = 1/n$, where n is the desired distance between shortcuts. Our goal was to achieve the same effect on the garbage collector, while reducing the synchronization required by the shared counter used in our deterministic variant. Measuring the idealized trace utilization, however, the results of the random variant were always less scalable than the deterministic version. In this chapter we provide the complete results of running the artificial benchmark with the random shortcut queue, compared with the original queue and the deterministic version of the shortcut queue.

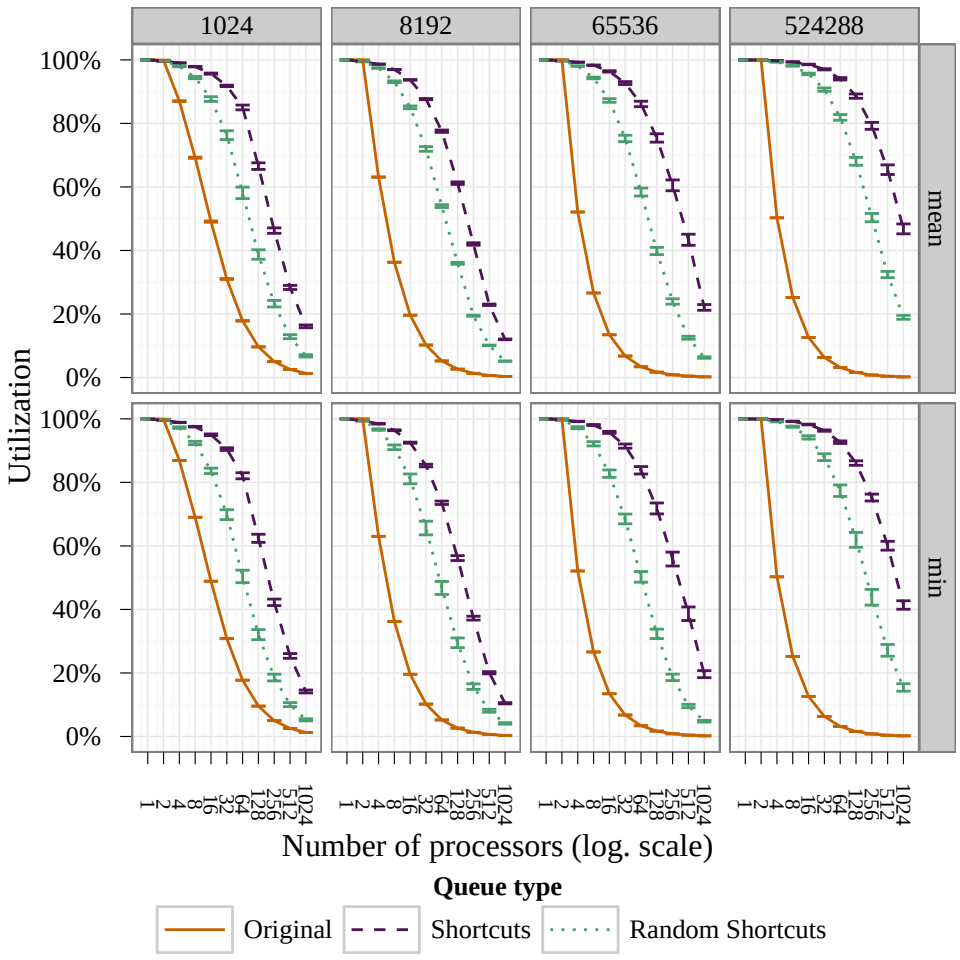

Figure 7.1: Idealized trace utilization for the artificial benchmark

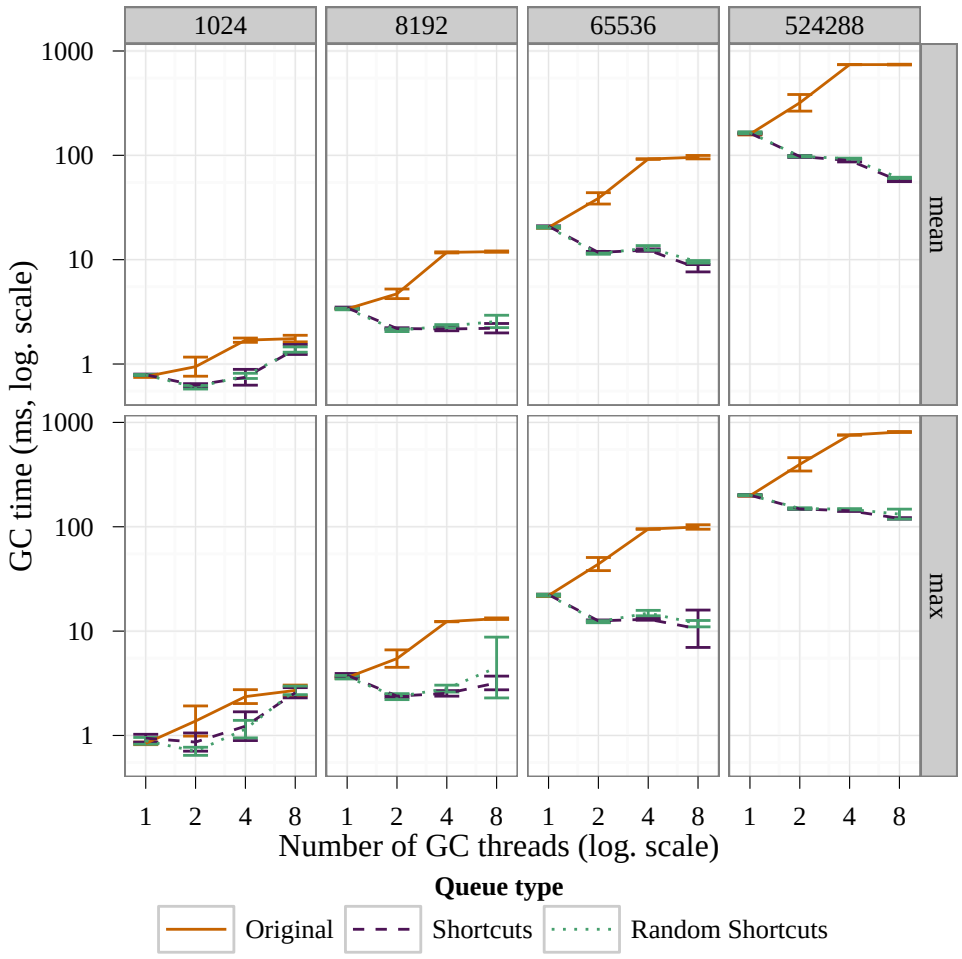

Figure 7.2: Artificial benchmark garbage collection time under the HotSpot JVM

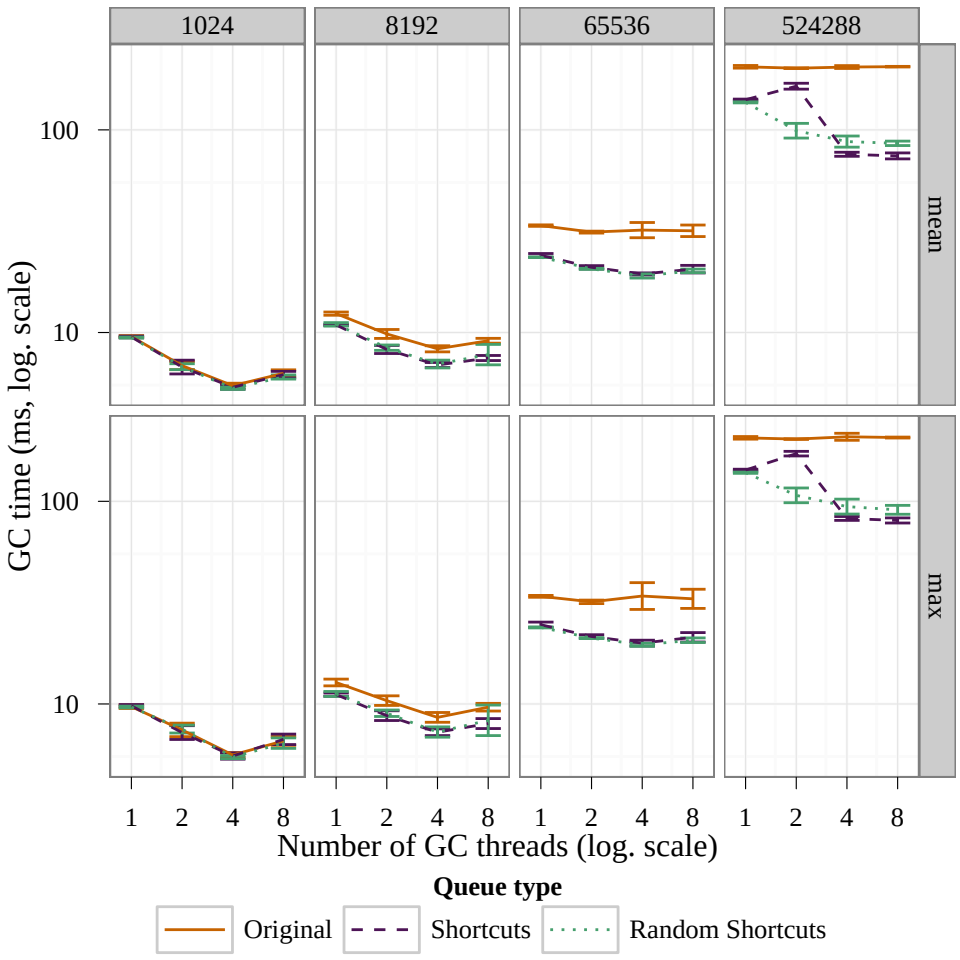

Figure 7.3: Artificial benchmark garbage collection time under IBM's JVM

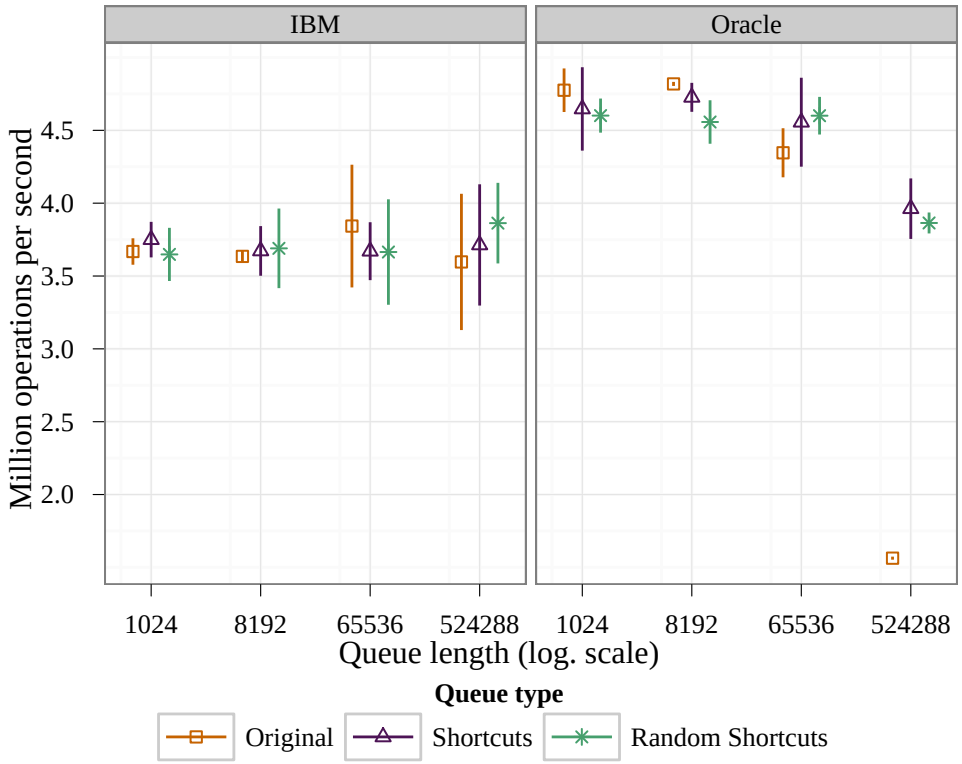

Figure 7.4: Artificial benchmark performance (queue operations per second) by JVM, queue length, and queue type.

Chapter 8

Conclusion and Future Directions

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 to replace the linked-lists with skip-lists and we also 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. The skip-list usage improved the heap shape of `bloat`, but added some overhead to the collection trace. For all other 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.

Bibliography

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005. doi:10.1147/sj.442.0399.

[2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03*, SIGPLAN Notices 38(11), pages 269–281. ACM, 2003. doi:10.1145/949305.949329.

[3] David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *PLDI '01*, pages 92–103. ACM, June 2001. doi:10.1145/378795.378819.

[4] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *TOPLAS*, 27(6):1097–1146, November 2005. doi:10.1145/1108970.1108972.

[5] Katherine Barabash and Erez Petrank. Tracing garbage collection on highly parallel platforms. In *ISMM '10*, pages 1–10. ACM, June 2010. doi:10.1145/1806651.1806653.

[6] Stephen M. Blackburn, Robin Garner, Chriss Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development

and analysis. In *OOPSLA '06*, SIGPLAN Notices 41(10). ACM, 2006. doi:10.1145/1167473.1167488.

[7] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI '91*, pages 157–164. ACM, 1991. doi:10.1145/113445.113459.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, Grenoble, France, November 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769.

[9] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI '01*, pages 125–136. ACM, June 2001. doi:10.1145/378795.378823.

[10] Cliff Click. Private communication.

[11] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978. doi:10.1145/359642.359655.

[12] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, pages 70–83. ACM, 1994. URL: ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz, doi:10.1145/174675.174673.

[13] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL '93*, pages 113–123. ACM, January 1993. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz, doi:10.1145/158511.158611.

[14] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In *ISMM '02*, pages 76–87. ACM, 2002. URL: http://www.cs.technion.ac.il/~erez/papers.html, doi:10.1145/512429.512439.

[15] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java.

In *ISMM '00*, pages 155–166. ACM, 2000. URL: http://www.cs.technion.ac.il/~erez/papers.html, doi:10.1145/362422.

[16] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *PLDI '00*, pages 274–284. ACM, 2000. URL: http://www.cs.technion.ac.il/~erez/papers.html, doi:10.1145/349299.349336.

[17] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *ICS*, November 1997. doi:10.1109/SC.1997.10059.

[18] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *IPDPS '01*, pages 43–48. IEEE Computer Society, 2001. URL: http://dl.acm.org/citation.cfm?id=645609.662496.

[19] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Reducing pause time of conservative collectors. In *ISMM '02*, pages 12–24. ACM, 2002. doi:10.1145/512429.512432.

[20] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01*, Monterey, CA, April 2001. USENIX. URL: http://www.usenix.org/events/jvm01/flood.html.

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

[22] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 48–57, Palo Alto, CA, June 2001. ACM. URL: ftp://ftp.cs.umass.edu/pub/osl/papers/jgrande-2001.ps.gz, doi:10.1145/376656.376810.

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

[24] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA '01*, pages 367–380. ACM,

2001. URL: http://www.cs.technion.ac.il/~erez/papers.html, doi:10.1145/504282.504309.

[25] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. In *ISMM '11*, pages 21–32. ACM, June 2011. doi:10.1145/1993478.

[26] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–275, New York, NY, USA, 1996. ACM. URL: http://doi.acm.org/10.1145/248052.248106, doi:http://doi.acm.org/10.1145/248052.248106.

[27] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *ISMM '00*, pages 143–154. ACM, 2000. doi:10.1145/362422.362480.

[28] William Pugh. A skip list cookbook. Technical Report UMIACS-TR-89-72.1, University of Maryland, College Park, MD, USA, 1990. URL: http://hdl.handle.net/1903/544.

[29] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *CGO '08*, pages 175–184, Boston, MA, April 2008. ACM. doi:10.1145/1356058.1356082.

[30] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006. doi:10.1145/1147954.1147958.

[31] Fridtjof Siebert. Limits of parallel marking collection. In *ISMM '08*, pages 21–29, Tucson, AZ, June 2008. ACM. doi:10.1145/1375634.1375638.

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

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

[34] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975. doi:10.1145/361002.361005.

[35] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976. doi:10.1145/360238.360247.

[36] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM '00*, pages 18–24. ACM, 2000. doi:10.1145/362422.362432.

[37] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05*. IEEE, 2005. URL: http://compilers.cs.ucla.edu/avrora/papers/avrora_ipsn2005.pdf.

חקר מבני נתונים היוצרים ערימה עמוקה

חגי ערן

חקר מבני נתונים היוצרים ערימה עמוקה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי המחשב

חגי ערן

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אייר ה'תשע"ב חיפה אפריל 2012

המחקר נעשה בהנחיית פרופ' חבר ארז פטרנק בפקולטה למדעי המחשב.

ברצוני להודות מקרב לב למנחה שלי, פרופ' חבר ארז פטרנק, על ההנחיה שלו. במהלך עבודתנו הוא אתגר אותי ונסך בי בטחון. האופטימיות שלו עזרה לי להמשיך גם כאשר דברים לא הסתדרו כפי שרצינו.

אני מודה לועדת הבוחנים שלי, ד"ר ערן יהב ופרופ' חבר עדית קידר, ולמבקרים אנונימיים. הערותיהם ושאלותיהם סייעו לשפר את איכות העבודה.

אני רוצה להודות גם למשפחתי ולקהילתי, על התמיכה והעידוד שנתנו לי.

לבסוף, אני מבקש להודות לאשתי שושן, על האהבה וההבנה שלה בתקופת לימודיי, על העניין שהביעה ועל ההשתתפות בחלק זה בחיי.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תקציר

בשנים האחרונות מערכות מחשבים מרובות־ליבות הפכו שכיחות, ומחשבי העתיד יהיו יותר ויותר מקביליים. מתכנתים נדרשים ללמוד טכניקות חדשות על מנת לנצל את מלוא הפוטנציאל של החומרה בת-ימינו ושל חומרת העתיד. נשאלת השאלה, האם מערכות זמן הריצה יוכלו גם הן לנצל ליבות מרובות באופן יעיל. בעבודתנו בחנו שאלה זו על מערכת ניהול הזכרון, ובפרט איסוף האשפה.

שפות מודרניות כגון Java ו-C# משתמשות באיסוף אשפה לניהול הזכרון. אף אל פי שקיימים אלגוריתמים רבים לאיסוף אשפה מקבילי, אלגוריתמים כאלה לא יצליחו לנצל ביעילות את הליבות הרבות אם צורת גרף האוביקטים של התכנית אינה מתאימה לסריקה מקבילית. לדוגמה, סריקה של רשימה מקושרת ארוכה היא סדרתית מטבעה, ולא ניתן לבצעה במקביל. חוק אמדל קובע שההאצה של תוכנית מקבילית מוגבלת על ידי הזמן הנדרש להרצת החלק הסדרתי בתכנית. במקרה הנ"ל, לא משנה עד כמה מקבילית תהיה הסריקה של שאר הערימה, המעבר על הרשימה ידרוש את מלוא הזמן שהיה נדרש לסרוק אותה סדרתית. קיומן של בעיות ביצועים כאלה הוא מטריד, כיוון שהמתכנת אינו מודע בהכרח שהבחירה שלו במבנה נתונים מסוים תפגע בביצועי חלקים של מערכת זמן־הריצה, והוא אינו מבין מדוע.

דרך מסוימת לבחון את התאמת צורת הערימה לסריקה מקבילית, היא להסתכל בעומק המקסימלי של הערימה. אלגוריתמים לאיסוף אשפה סורקים את הערימה החל מקבוצת מצביעים הנקראת שורשים, כך שכל אוביקט שיש לו מסלול בגרף האוביקטים מאחד השורשים, מוגדר כאוביקט "חי", והאלגוריתם מפנה את שאר האוביקטים. אורך המסלול הקצר ביותר משורש לאוביקט מוגדר כעומק של האוביקט, והעומק המקסימלי של אוביקט כלשהו מוגדר כעומק הערימה כולה. קל לראות שסריקה של הערימה תדרוש לכל הפחות מספר צעדים כעומק הערימה. יחד עם זאת, יתכן שבערימה יש מספר מספיק של אוביקטים בעומק המקסימלי, כך שתמצא מספיק עבודת סריקה מקבילית לכל הליבות במערכת. למשל, מערכת עם k ליבות תוכל לבצע סריקה של k רשימות מקושרת במקביל, גם אם הן מאוד ארוכות.

מדד טוב יותר להתאמת הערימה לסריקה מקבילית הוא מדד הניצולת בסריקה אידיאלית (Idealized Trace Utilization) שהוצע על ידי [5]. מדד זה מנסה להעריך בקירוב את ניצולת המעבדים בזמן סריקת הערימה הנתונה, תוך כדי הנחה של חלוקת עומס מושלמת בין המעבדים, וסריקה מיידית של אוביקטים. המדד גם מניח סריקת BFS, כיוון שהיא מכוונת למקבול רב יותר. מדד הניצולת בסריקה אידיאלית מחושב על ערימה נתונה, עם מספר נתון של מעבדים מדומים, על ידי חישוב מספר המחזורים הנדרשים לסריקת הערימה, כך שבכל מחזור כל מעבד לוקח אוביקט בודד מתור BFS גלובלי, סורק את אותו אוביקט, ומכניס את כל האוביקטים המוצבעים שעדיין לא נסרקו חזרה לתור.

ניצולת הסריקה האידיאלית היא אם כן מנת מספר האוביקטים הכולל, במספר המעבדים ובמספר המחזורים. אנו משתמשים בו על מנת לקרב את אחוז מחזורי המעבדים המנוצלים, במקרה הטוב ביותר כאשר במערכת אין בעיות של חלוקת עומס, החטאות מטמון, או סינכרון. כאשר הניצולת נמוכה, אנו מסיקים שלסריקה מקבילית של הערימה יהיה קושי לנצל מספר גדול של מעבדים.

בעבודה זו התחלנו בחקירת תכניות מבחן מקובלות ל-Java, אשר בבחינת מדד הניצולת בסריקה אידיאלית הפגינו תוצאות נמוכות. ניסינו להבין מהם מבני הנתונים הגורמים לצורת ערימה בעייתית, וכיצד התכניות משתמשות בהם. חקרנו תכניות מבחן מתוך האוספים DaCapo משנת 2006 ומשנת 2009, ומתוך SPECjvm98 ו-SPECjvm2008. נראה שהגורמים העיקריים לערימות בעלות מבנה עמוק בתוכניות שבחנו הם רשימות מקושרות ותורים. חלק מן התכניות השתמשו בתורים המאפשרים עבודה מקבילית של מספר חוטים על מבנה הנתונים, אך חלקם השתמשו ברשימות מקושרות באופן שמניח אך ורק עבודה עם חוט יחיד. במבנים אלה היה לנו מעט עניין לעסוק, כיוון שגם אם נצליח למקבל את סריקת הרשימה, אם התכנית עצמה פועלת באופן סדרתי היא לא תוכל לנצל את מלוא יכולת המערכת. לפיכך בחרנו להתמקד בתור, שהשימוש בו מתאים גם למערכות מקבילות. אף על פי כן חקרנו בעבודה זו גם חלופות לרשימות אלה, כמו שימוש ברשימת דילוגים (skip list) או ברשימה שאיבריה הם מערכים.

עבור התור, הצענו מימוש חלופי לתור המקבילי של Java, ConcurrentLinkedQueue, אשר כולל מצביעי עזר נסתרים. התכנית אינה משתמשת במצביעים אלה, אך סריקת איסוף האשפה נעזרת בהם לחלוקת העבודה בין הליבות. התור החדש כולל שדה קיצור־דרך בכל צומת ברשימה, אשר מצביע לצומת הממוקם n צעדים קדימה בהמשך הרשימה. מצביעים אלה יוצרים רשימה מקושרת נוספת של קיצורי הדרך. כדי להחליט מתי להוסיף קיצור דרך כזה, התור מחזיק גם שדה גלובלי עם מספר ההכנסות לתור שנעשו עד כה, ומצביע לצומת האחרון שנוסף עם קיצור דרך. כאשר מכניסים צומת חדש לתור, בודקים אם ערך מונה ההכנסות מתחלק ב-n. אם כן, נדרש קיצור דרך חדש, והאלגוריתם מנסה להכניס את הצומת החדש לרשימת קיצורי הדרך. המימוש המקורי של התור הוא חסר נעילות (lock free) והתור החדש שומר על תכונה זו.

מימשנו את התור החדש ובדקנו אותו עם מספר תוכניות מבחן, על מנת להראות שהתקורה שלו נמוכה, ושהוא משפר את ניצול המעבדים במערכות מקביליות. התוצאות הראה שהתור החדש משפר משמעותית את מדד הניצולת בסריקה אידיאלית, ובמדידת זמן הריצה של התכניות לא התגלה הבדל משמעותי עם התור החדש, כך שהתקורה שלו אינו משמעותית ביחס לתוכנית. בחנו גם את זמן הריצה של איסוף האשפה, אך למעט מספר תכניות שבהן התגלה שיפור בזמן הריצה, ברוב התכניות לא התגלה הבדל משמעותי בזמן הריצה של איסוף האשפה. ככל הנראה הסיבה לכך היא שמערכת המבחן שלנו כללה שמונה ליבות בלבד, בעוד שנדרש מספר רב יותר של ליבות על מנת לראות את תופעות הניצול הנמוך כפי שחזה מדד ניצולת הסריקה האידיאלית.

לסיכום, ראינו שצורת הערימה עשויה לפגום ביעילות איסוף האשפה, וביכולתו לנצל את הפוטנציאל הגלום במערכות מרובות מעבדים. בעבודה זו הרחבנו את אוסף התוכניות עבורן נבחנה בעיה זו. ניתחנו את התוכניות שהציגו צורת ערימה שאינה מתאימה למקבול, ומצאנו שהבעיה העיקרית היא בשימוש של תכניות בתורים וברשימות מקושרות. הצענו מימוש חלופי לתור, בעל תקורה נמוכה, שהשיג שיפור משמעותי עבור איסוף אשפה מקבילי.