Scalable Garbage Collection on Highly Parallel Platforms

Katherine Barabash
Scalable Garbage Collection on Highly Parallel Platforms

Research Thesis

In Partial Fulfillment of The Requirements
for the Degree of Master of Science
in Computer Science

Katherine Barabash

Submitted to the Senate of the Technion — Israel Institute of Technology
Heshvan 5771 Haifa October 2010
The Research Thesis was Done under the Supervision of Associate Professor Erez Petrank in the Department of Computer Science

I am very grateful to Assoc. Prof. Erez Petrank for his excellent guidance during this work. Erez knew exactly how to find a delicate balance between being there when needed, providing encouragement and direction, and allowing me to develop this work in my own way.

I wish to express gratitude to anonymous conference reviewers and to my thesis examiners, Assoc. Prof. Roy Friedman, Dr. Hillel Kolodner, and Prof. Assaf Shuster. Their insightful questions and comments helped improving the final presentation of this work.

I thank the IBM Haifa Research Lab management and HR team for encouraging me to take up this challenge and for supporting my M.Sc. studies.

I also thank my family for their patience and support during the period I was invested in my studies and research more than in their day-to-day lives. I am certainly going to make up for this now.
Contents

Abstract

List of Symbols

1 Introduction
  1.1 Background ........................................ 4
  1.1.1 The Garbage Collection .......................... 5
  1.1.2 The Manycore Challenge ......................... 7
  1.2 Objectives .......................................... 8
  1.3 Approach ............................................ 9
  1.4 Related Work ........................................ 10
  1.5 Organization ........................................ 12

2 Evaluating the Object-Graph Scalability .............. 13
  2.1 Object-Graph Depth ................................. 13
  2.2 Object-Graph Shape .................................. 17
  2.3 Idealized Trace Utilization Measure ................ 19
  2.4 Idealized Trace Utilization Measurements .......... 24

3 Improving the Object-Graph Scalability ............... 26
  3.1 Adding Shortcut References ....................... 26
  3.1.1 Shortcut Adding Algorithm ..................... 28
  3.1.2 Possible Optimization .......................... 31
  3.1.3 Performance Considerations .................... 31
  3.2 Tracing Randomly in Parallel ...................... 32
  3.2.1 Colors Table ..................................... 34
  3.2.2 Main Trace ....................................... 34
  3.2.3 Helper Trace ..................................... 34
  3.2.4 Trace Completion ................................ 35
  3.2.5 Random choices with Filters and Biases ........ 35
  3.2.6 Performance Considerations .................... 36
4 Implementation

4.1 Instruments and Equipment ........................................ 37
  4.1.1 The Hardware and OS Platform ................................ 37
  4.1.2 The Language Platform ........................................ 37
  4.1.3 The Benchmarks .............................................. 38

4.2 Prototypes ............................................................ 42
  4.2.1 Modifying the Trace ........................................... 42
  4.2.2 Computing the Idealized Trace Utilization ................. 43
  4.2.3 Adding Collection Phases ..................................... 44
  4.2.4 Modifying the Object Headers ................................. 48
  4.2.5 Adding Shortcut References .................................... 48
  4.2.6 Tracing Randomly in Parallel ................................. 49

5 Results ................................................................. 51
  5.1 Adding Shortcut References ....................................... 51
  5.2 Tracing Randomly in Parallel ..................................... 55

6 Conclusion .............................................................. 61

A Computer Programs .................................................... 62
  A.1 Adding Shortcut References Code ................................ 62
  A.2 Tracing Randomly in Parallel Code ............................... 66

References .............................................................. 70

Hebrew Abstract .......................................................... i
List of Figures

2.1 Live objects depth distribution for Java benchmarks ........................ 18
2.2 Worst case idealized trace utilization ........................................... 23
2.3 Average idealized trace utilization .............................................. 24

3.1 An example of a linked list with shortcuts added ............................... 29
3.2 An example of a red trace, which is hit by the main trace .................. 33

5.1 Worst case object-graph trace utilization before and after adding shortcuts for *jython* .............................................. 53
5.2 Worst case and average object-graph trace utilization before and after adding shortcuts for *mtrt* ............................................ 53
5.3 Worst case and average object-graph trace utilization before and after adding shortcuts for *xalan* ............................................ 53
5.4 Worst case and average object-graph trace utilization before and after adding shortcuts for *bloat* ............................................ 54
5.5 Worst case object-graph trace utilization before and after adding shortcuts for *javac* and *pmd* ............................................ 55
5.6 The change in idealized trace utilization when using random trace for *mtrt* (no filter) .......................................................... 56
5.7 The change in idealized trace utilization when using random trace for *mtrt*, with the filter ......................................................... 57
5.8 The change in idealized trace utilization when using random trace for *javac*, no filter .............................................................. 58
5.9 The change in idealized trace utilization when using random trace for *javac*, with the filter .......................................................... 58
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Object graph depths for SPECjvm98 benchmarks when GC cycle is forced every 32K of allocations</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Object graph depths for Dacapo benchmarks when GC cycle is forced every 1M of allocations</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Object graph depths for Java benchmarks for regular runs</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>The summary of live objects distribution on different distances from roots</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Benchmarks’ memory usage properties</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Properties of the random trace with no filter</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Properties of the random trace with the random choices filter</td>
<td>60</td>
</tr>
</tbody>
</table>
Abstract

The computing landscape is changing rapidly in recent years. On the one hand, the pervasiveness of multiprocessor and multicore hardware requires the software to be able to take advantage of the increasingly massive hardware parallelism. On the other hand, the growing complexity of the modern software application domains makes runtime language environments more popular as a major software development tool. It is predicted that these trends will progress: we can expect runtime language systems having to sustain highly complex and memory demanding workloads on servers with hundreds of processor cores in the near future.

An important related question is whether a garbage collector, being a major part of the modern runtime language environment, is able to run efficiently on highly parallel hardware platforms of tomorrow. Our hypothesis was that the structure of the live objects graph in a garbage collected heap can influence the ability of a collector to scale. In particular, certain, sequential in nature, patterns in the live objects graph structure can prevent the tracing garbage collector from scaling the important collection phase, that is – tracing through the object graph.

In this work, we have studied the above hypothesis for the Java programming language using Jikes Research Java Virtual Machine as a platform and a set of standard technology Java benchmarks as applications.

First, we have examined the object graphs created by all the considered applications. We have measured object graph shape properties such as depth and width and devised a measure that can be used to evaluate the amount of tracing scalability the object graph allows for. We have named this measure the *idealized trace utilization* measure and have used it to evaluate applications in terms of their ability to sustain parallel tracing without causing it to become serial. Some of the applications we have investigated, exhibited problematic object graph shapes. These same applications scored poor idealized trace utilization measure readings.

Next, we have studied two approaches for alleviating the scalability problems caused by the problematic object graph shapes. The first approach is modifying the application’s object graph shape by adding new inter object references, invisible to the application but useful for the tracing threads. The second approach is modifying the tracing process by allowing idle tracer threads to pick additional tracing roots in attempt to obtain more tracing work. For each approach, we have designed and
prototyped one specific solution. We have evaluated each solution’s influence on the object graph shape of all the applications in our benchmarks set.

In this thesis, we present and analyze the results obtained by both approaches. More promising results were achieved by the method of adding new references to the application’s object graph.
# List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>BHT</td>
<td>Branch History Table</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions per Cycle</td>
</tr>
<tr>
<td>ITU</td>
<td>Idealized Trace Utilization</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MMTk</td>
<td>Memory Management Toolkit for Jikes RVM</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RVM</td>
<td>Research Virtual Machine for Java</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

During recent years, we have witnessed a fundamental change in how computer productivity is approached. The exponential growth of the processor speeds we have been used to for several decades has come to an end and efforts to speed up serial computation have been abandoned in favor of increasing hardware parallelism. Dual core desktops are now standard for home and office, computers installed in server farms have an increasing number of processors and cores, and even embedded systems have started to enjoy multicore designs. As was predicted by researchers and technologists [57], this new approach to hardware productivity has brought about a fundamental change in the software landscape as well. Sequential programs can no longer benefit substantially just by being run on a newer hardware. In order to harvest the benefits of new parallel hardware, new parallel software must be created.

Runtime programming languages such as Java and C# are becoming the main vehicle for realizing large software projects. They provide built-in security, threading support, impressive multi purpose standard class libraries, and last, but not least, the dynamic memory management and garbage collection (GC). In order to maintain the advantages of runtime languages, applications written in these languages must be efficient and scale well on modern and future platforms. In particular, runtime language systems and garbage collection algorithms must be made adequately scalable to support efficient execution on future hardware platforms.

In this work, we claim that multiprocessor garbage collection techniques available today may fail to fit highly parallel platforms of tomorrow; we raise and study the question of relevance of the object graph shape properties to the garbage collector scalability.
1.1.1 The Garbage Collection

Dynamic memory management and automatic garbage collection were introduced in 1960s by McCarthy for the LISP language system [40]. Automatic garbage collection is a service that allows programmers to allocate objects and manipulate them without a need to worry about their reclamation. Dynamic memory management system takes care of discovering objects that cannot be used by a program (the garbage objects) and returns their memory to the allocation pool. Back in 1960s, garbage collection was causing significant runtime overheads and very long program pauses. On the positive side, garbage collection relieved programmers from manual memory management tasks and greatly increased programmer productivity and software quality. Although for many years dynamic memory management was considered inefficient and not suitable for real industrial workloads, its benefits were continuously driving the research and the development of new algorithms and techniques. Today, garbage collected systems are widespread (Java, C♯, Ruby, Javascript, etc) and their performance begins to match the performance of manually managed systems (C, C++, etc) [52].

There are two basic approaches to garbage collection – live objects’ reference counting and tracing objects reachable from program’s roots. These approaches gave rise to three popular garbage collection algorithms – reference counting, copying, and mark sweep. All the algorithms require some form of object graph traversal as described below.

Reference counting collectors, introduced by Collins in 1960 [23], continuously keep track of the amount of references to every heap object. When the reference counter for some object becomes zero, the object is not reachable by the program and can be reclaimed. As part of object reclamation, there is a need to decrement reference counters of objects referenced by the reclaimed object. Some of these objects may have to be reclaimed in the process so that their referents’ reference counters must be updated as well. This process can lead to a traversal of all the objects in the reclaimed object’s reachability tree.

Several years after its invention, the basic reference counting algorithm was found unable to collect garbage cycles [45] and was augmented with the cycle collection techniques. Cycle collection algorithms involve multiple traversals of object graph areas suspected of containing garbage cycles.

Tracing collectors operate in cycles that are triggered by a trigger, usually based on the heap utilization properties. When triggered, the collector computes and marks all the objects referenced from the program stacks and global variables. These objects are considered roots of the current collection cycle. All the objects reachable from roots can be used by the program and thus are conservatively assumed to be live; all the other objects cannot be used by a program anymore...
and thus are reclaimed and their memory is used for allocation till the next collection cycle is triggered.

**Copying** collectors, introduced by Cheney [19] in 1970, divide the available heap space into two parts. Objects are allocated in one part and when it becomes full, the collection cycle is triggered. During the collection cycle, the live objects graph is traversed starting from root objects and every discovered object is copied to the empty part of the heap. Roots and heap objects are updated to refer to the new copies of the relocated heap objects. At the end of the traversal, one part of the heap contains only garbage and can be emptied and the other part contains all the program’s live objects. At this point, the program is resumed and new objects are allocated in the heap part containing live objects.

Main advantage of copying collectors is that after every cycle, live objects are compacted at the beginning of the allocation area. The disadvantages are: copying objects can be costly; there is a need to fix references pointing to the copied objects; memory requirements are doubled.

**Mark and Sweep** collectors, introduced by McCarthy [46] in 1960, trigger collection cycle when the amount of memory available for allocation reaches the predefined lower limit. There are two phases to the collection cycle – *mark phase* and *sweep phase*. During the mark phase the collector traverses the heap starting from the roots and marking all the discovered objects as live. At the end of the mark phase, all the live objects are marked and all the unmarked objects are unreachable. Unmarked objects are returned to the allocation pools by the sweep phase.

The advantages of mark and sweep collectors are that all the memory can be used for allocation and there is no need for costly object copying and pointer updates. The disadvantage is that the heap can become fragmented after a few cycles making the allocation more complex and costly.

For more details on classical garbage collection algorithms and their variations, see [40].

As we can see from the above, all the garbage collection algorithms require some sort of heap objects traversal. Reference counting collectors trace dead objects starting from the object being reclaimed and during garbage cycles collection. Tracing collectors periodically trace all the live objects in the heap starting from the program roots. The object traversal process is iterative (or recursive) and has to keep track of objects that were already discovered but not yet scanned for their referents. Garbage collectors typically maintain stacks or queues for discovered objects tracking, thus creating a depth first search (DFS) or a breadth first search (BFS) object graph traversal. A DFS traversal is usually considered more cache friendly [53]. A BFS
traversal is more scalable in the presence of multiple collector threads because more paths are discovered earlier on and better load balancing is possible.

Basic garbage collection algorithms were greatly enhanced since their introduction in 1960s. The research was driven mainly by the performance goals – shorter pause times, better program performance (running time, response time, and throughput), and smaller memory footprint. Incremental and generational collection techniques were invented to cope with increasing heap sizes and to reduce pause times [44, 59]. Concurrent and on-the-fly collection techniques helped to fully exploit the computing power of uniprocessor and a small number of additional processors and to further reduce pause times trading this reduction for greater runtime overhead [55, 56, 25, 27, 26, 28, 29, 51, 39, 43, 11, 10]. With the appearance of hyper threading, multi processor and multi core hardware, the importance of these techniques has become even greater. Parallel garbage collection algorithms [42, 17, 30, 20, 32, 13, 60] were introduced to answer the challenges of supporting parallel execution paradigm. Recently, garbage collected languages have become widespread and efficient enough to be considered as candidate platform for real time systems and applications. As a result, the need for real time garbage collection techniques has emerged [12, 22, 49].

1.1.2 The Manycore Challenge

For about 15 years, starting with the appearance of the first RISC processors in 1986, microprocessors have been improving in overall performance at a rate of approximately 50-60% per year [5]. These substantial performance improvements have been obtained mainly from two sources. First, designers have been increasing clock rates, both by scaling technology and by reducing the number of levels of logic per cycle. Second, designers have been exploiting the increasing number of transistors on a chip, plus improvements in compiler technology, to improve instruction throughput (IPC). In addition, performance improvements were gained by allowing larger and cheaper processor memories that have reduced the amount of page faults during program execution. Although sometimes the effect of one of these trends was emphasized over the other, both clock rates and IPC have been improving consistently. For a long time, it was expected that these trends would continue into the future, maintaining the same rate. The prediction of the International Technology Roadmap for Semiconductors in 2005 was that clock rates should have exceeded 10GHz in 2008, reaching 15GHz in 2010. The major architectural problem identified back then, was the gap between the rate of improvement in the processor speed and the rate of improvement of the memory latencies. As a result, most of the research in computer architecture at that period was involved with resolving this ‘memory wall’ issue.

But the unexpected happened – at some point designers could not speed up the processor clocks further because linear increase in clock frequency creates a cubic increase in power consumption [48]. The amount of power generated by the fastest
processors became so significant that it was very hard and expensive to remove. The prediction of the International Technology Roadmap for Semiconductors was updated in 2007 to aim to about 8GHz in 2013. Even this conservative prediction failed to account for the technological difficulties that were encountered and, as a result, the processor clock rates ceased increasing and have been basically flat for about 5 years. For example, clock rates of Intel processors today are far below even the conservative 2007 prediction. Not only the processor speed stopped to increase, but the pipeline technologies started to hit the limits of practicality in branch prediction and speculative execution. As a result, the IPC improvements have slowed down as well. According to Agarval et al [5], no scaling strategy permits annual performance improvements of better than 12.5% for the serial computation.

As it has became either impossible or impractical to make the serial computation any faster, processor manufacturers decided to place several cores on the die as a way to achieve better overall performance. From the computer architecture perspective, it is expected that the number of cores per processor chip will double with each technology generation or every two years, while the frequency of the cores will remain the same. The question is whether software will be able to pick up the challenge and to translate increasing hardware parallelism into increasing overall performance. If not, there will be no incentive to buy newer hardware and the computer industry will begin to stagnate, turning from a growth industry into a replacement industry [8].

There is a large amount of ongoing research addressing the manycore challenge these days [48]. This work joins this effort in an attempt to prepare garbage collection techniques for the hardware of tomorrow.

1.2 Objectives

In this work, we study the scalability of the object graph tracing and how it is influenced by the object graph shape properties. We concentrate on tracing garbage collection algorithms where the tracing phase is more important. In particular, we investigate the mark and sweep garbage collection algorithms which are most often employed in today’s large systems. Modern mark and sweep collectors are generational, incremental, concurrent, mostly-concurrent or on-the-fly. Some collectors are parallel, so that garbage collection work is performed by several processors simultaneously. Modern mark and sweep collectors are very efficient, feature low runtime overheads and short pauses. However, there is a question of whether they are capable of scaling to a very large number of processors. During the mark sweep collection cycle, the garbage collector has to: 1) scan application threads’ stacks and global variables for roots; 2) trace through all the live objects from roots; 3) reclaim storage occupied by dead objects. To make the collector parallel, this work has to be efficiently distributed among the collector threads. The challenge is to distribute the
work evenly and to achieve this without much synchronization overhead. It is relatively straightforward to parallelize the root scanning and the reclamation phases. Unfortunately, it is not so for the tracing phase. The first parallel garbage collection algorithm [36] did not employ any load balancing and its scalability was poor because of imbalances in work distribution. Later algorithms employed load balancing techniques during trace: work sharing [20], work stealing [30, 32, 13], work packets [13], task pushing [60]. Fine-grained load balancing helped to achieve almost linear scalability on up to 16 processors on SPECjbb benchmark [9], while more massive parallelism was not fully explored yet. Other techniques aimed at improving the scalability of parallel garbage collection algorithms are hardware assisted barriers [22], lock freedom [38, 37, 49, 34], and transactional memory [35, 47]. Now, when the number of available cores is rapidly growing, garbage collectors must become massively parallel to cope with high allocation rates and huge heaps of modern massively parallel applications. Garbage collection pauses must be made not simply very short but predictably short in order for garbage collected languages to be acceptable in real-time systems.

In this work, we claim that even if the garbage collection tracing work was balanced perfectly, the shape of the object-graph is important as it can foil the tracing scalability [17, 30, 32, 13, 60]. Imagine a program that employs a large linked list of heap objects. A traversal of such a list is sequential in nature and cannot benefit from having more than one tracing processor. Other less extreme object-graph shapes may be detrimental to parallel traversal as well. A question that naturally arises is whether this problem of heap shapes that may fail tracing scalability actually exists in typical programs. And if the problem does exist, then what can we do about it? Namely, is there a way to ameliorate this problem and to facilitate scalability of the garbage collector for such programs? If we cannot solve this problem, then future scalability of Java and C# runtimes becomes questionable.

This work initiates a rigorous investigation of the heap tracing scalability issue and how the object graph properties influence the scalability of the object graph tracing.

1.3 Approach

We start by studying the object graph properties of a set of standard Java benchmarks (the SPECjvm98 benchmark suite, the SPECjbb2005 benchmark, and the DaCapo benchmark suite [4, 14]) in order to look for problematic data structures and object graph shapes.

We proceed by proposing an idealized trace utilization measure that, given an object-graph shape, evaluates the amount of parallelism the object-graph enables. Our measure is highly intuitive, in the sense that it simulates a clean parallel trace of this object-graph shape for a given number of processors.
Next, we use the idealized trace utilization measure to evaluate the object-graph shapes of standard Java benchmarks. Our measurements show that non-scalable object-graph shapes exist for some of the investigated benchmarks.

We then propose, prototype, and evaluate two solutions to this problem. Our solutions attempt to add functionality to the collector in order to ameliorate non-scalability of object-graph shapes. The first solution is to add pointers to the headers of objects, which artificially modify the object-graph shape and make it look more scalable to the tracer threads. The second solution lets additional garbage collection threads run on idle processors. These auxiliary collector threads pick objects at random in the heap and trace from them as if they were alive. At a later stage, it is determined whether the set of objects traced by each thread is reachable or not. The additional tracing can happen concurrently on many processors even if it is not yet clear which objects are alive, and thus this method entails high concurrency even with linked lists.

We have implemented prototypes of both solutions on Jikes RVM [7] using the MMTk [3, 16], the garbage collection framework. Section 4 describes the implementation details and contains relevant architecture and implementation information about the platform.

We have evaluated our prototype solutions using the methodology based on idealized trace utilization measure and standard Java benchmarks. We describe the evaluation methodology and present and discuss the results in Section 5.

1.4 Related Work

Researches working on parallel garbage collection algorithms have noted [17] that long linked lists of objects can be a source of load imbalance and poor scalability for parallel objects graph tracing. Developers of load balancing algorithms for parallel garbage collection trace [30, 32, 13, 60] either explicitly state that long linked lists present a scalability problem or implicitly assume that such structures are rare or non-existent.

Recently, a study of the relationship between the object graph depth and the scalability of parallel trace was reported by Siebert [54]. This study reports that there exist very deep data structures in several SPECjvm98 benchmarks and makes theoretical prediction on how these layouts can influence the scalability of garbage collection tracing. The author concludes that in order to be able to predict the duration of pauses caused by garbage collection tracing, it is required that the application writer declares what the possible maximal objects graph depth is.

Another relevant study trying to predict the scalability of parallel garbage collection tracing was performed back in 2001 by Endo et al [31]. There the authors state that scalability of parallel garbage collection trace can be realistically predicted
by taking into account both the object graph layout and the cache coherency issues. While we agree that both factors are important, we study only the first of these factors. In contrast with Siebert, we do not only relate the influence of the objects graph depth on the trace scalability and quantify it, but also investigate the algorithmic ways to improve the trace scalability in face of the problematic object graph layouts.

Click [21] proposed the idea of using idle processors to randomly trace objects and aid the trace of non-scalable heap shapes. As far as we know his proposal was not implemented, nor investigated prior to this work. Also, as far as we know, the method of adding shortcuts was not previously proposed.

Jump and McKinly [41] use dynamic shape analysis to characterize data structures in the heap and to identify recursive data structures. Their analysis is rather effective and efficient, incurring 4-8% runtime overhead. The purpose of the analysis is to compute dynamic degree metrics of the heap data structures during ‘good’ runs in order to be able to track violations of these metrics in the presence of errors. Thus, that work is similar to this one in that it studies the shapes of heap data structures, while being different in its methods and purposes. Jump and McKinly report that in the SpecJVM and DaCapo benchmarks, 91% of all the objects are part of recursive data structures with 67% of these objects coming from library implementations and 33% from custom implementations; sizes and depths of structure instances are not studied.

Data structures are used by programmers to keep and operate program’s data. Different data structures have different performance characteristics and are chosen based on these performance properties, well studied for the serial computation model. When parallel computation became widespread, these performance properties have been revisited [58] and two major sources of performance degradation were identified: locks contention required to correctly perform data structure operations and cache contention resulting from concurrent access to the same memory areas. To address the first problem, concurrency friendly lock free and wait free data structure designs have been introduced. Starting with version 1.5, Java contains new utility package with the concurrency friendly implementations of the common data structures such as lists, hash maps, etc. To address the second problem, hardware designers work on creating cache designs suitable for parallel data structure operations. In this work we raise the third problem that is located in the space of the managed runtime, specifically the tracing garbage collector. We initiate the investigation of how the existing data structure designs influence the garbage collector scalability. The ultimate outcome of such investigation can be new garbage collection techniques that can effectively scale with the existing data structure designs or new data structure designs suitable for the environments with parallel tracing garbage collectors.
1.5 Organization

In Section 2.1 we define and study heap depths of standard Java benchmarks. In Section 2 we describe the idealized trace utilization measure we have devised to quantify object-graph scalability. In Section 2.4 we present the idealized trace utilization measurements for a variety of widespread Java benchmarks, and demonstrate that some of these benchmarks generate object-graphs with poor scalability. In Section 3 we describe two possible solutions. Implementations are discussed in Section 4. We present and discuss the results in Section 5. We conclude in Section 6.
Chapter 2

Evaluating the Object-Graph Scalability

2.1 Object-Graph Depth

As described in Section 1.1.1, tracing garbage collectors must traverse all the heap objects reachable from a well-defined set of pointers called roots [40].

Each reachable object in the heap has one or more paths of pointers starting from some root object and leading to it. The length of the shortest such path defines the depth of the object or its distance from the roots. The depth of the entire live objects graph in the heap is defined as the maximum over the depths of all the reachable objects. Note that in order to discover an object of depth $d$ during a heap trace, at least $d$ pointers must be dereferenced sequentially. Therefore, deep objects are detrimental to the scalability of the trace. Even assuming a clean execution where each object is traced in a single clock tick\(^1\), it still holds that at least $\max\{L/P, D\}$ clock ticks will be required by $P$ processors to trace a heap with $L$ live objects and depth $D$. This is the reason we start investigating the object graph scalability properties from evaluating the object graph depth.

We have modified the mark and sweep stop-the-world garbage collector of Jikes RVM to measure the live object-graph depths during the trace. For this, the existing parallel DFS traversal was replaced by a single threaded BFS traversal. We have run this modified JVM for all benchmarks in the following benchmarks suites: SPECjvm98 [4], SPECjbb2005 [4], and DaCapo [1]. In order to capture dynamic changes in the object graph depth and to be able to register the maximal depth existing during the run, we have triggered a garbage collection very frequently during the test runs. Table 2.1 presents object graph depths for SPECjvm98 benchmarks when garbage collection is forced after every 32 KBytes of allocation. Table 2.2 presents

\(^1\)We use the term clock tick to denote the time it takes to execute a single computation step.
Table 2.1: Number of GC cycles, average number of live heap objects after GC, and maximal and average object-graph depths for the SPECjvm98 benchmarks when GC cycle is forced every 32K of allocations.

<table>
<thead>
<tr>
<th>name</th>
<th>cycles</th>
<th>live objects</th>
<th>max depth</th>
<th>avg depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>17</td>
<td>88,290</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>compress</td>
<td>957</td>
<td>127,310</td>
<td>38</td>
<td>27</td>
</tr>
<tr>
<td>db</td>
<td>1,463</td>
<td>354,494</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>jack</td>
<td>1,667</td>
<td>111,526</td>
<td>58</td>
<td>33</td>
</tr>
<tr>
<td>javac</td>
<td>1,129</td>
<td>310,262</td>
<td>1,235</td>
<td>234</td>
</tr>
<tr>
<td>jess</td>
<td>957</td>
<td>127,310</td>
<td>38</td>
<td>27</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>101</td>
<td>96,174</td>
<td>36</td>
<td>20</td>
</tr>
<tr>
<td>mtrt</td>
<td>1,345</td>
<td>335,557</td>
<td>1,419</td>
<td>1,353</td>
</tr>
</tbody>
</table>

Table 2.2: Number of GC cycles, average number of live heap objects after GC, and maximal and average object-graph depths for the Dacapo benchmarks when GC cycle is forced every 1M of allocations.

<table>
<thead>
<tr>
<th>name</th>
<th>cycles</th>
<th>live objects</th>
<th>max depth</th>
<th>avg depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>305</td>
<td>241,255</td>
<td>63</td>
<td>25</td>
</tr>
<tr>
<td>bloat</td>
<td>795</td>
<td>378,352</td>
<td>2,641</td>
<td>424</td>
</tr>
<tr>
<td>hsqldb</td>
<td>220</td>
<td>3,486,521</td>
<td>38</td>
<td>25</td>
</tr>
<tr>
<td>jython</td>
<td>1,861</td>
<td>529,216</td>
<td>128</td>
<td>124</td>
</tr>
<tr>
<td>lusearch</td>
<td>2,079</td>
<td>251,038</td>
<td>38</td>
<td>14</td>
</tr>
<tr>
<td>pmd</td>
<td>631</td>
<td>686,525</td>
<td>45,754</td>
<td>1,143</td>
</tr>
<tr>
<td>xalan</td>
<td>1,165</td>
<td>344,150</td>
<td>857</td>
<td>444</td>
</tr>
</tbody>
</table>
object graph dephs for Dacapo benchmarks when garbage collection is forced after every Megabyte of allocation. In addition, the tables show the number of GC cycles performed during the run and the average number of live heap objects after the collection cycle.

This step gave us a first glimpse into the existence of deep live object graphs in typical benchmarks, and furthermore, on how object graph depth changes during the program execution. It turned out that several benchmarks (javac, raytrace and mtrt of SPECjvm98 and bloat, pmd and xalan of Dacapo) exhibit deep and narrow object-graphs. We note that these results are consistent with the partial information provided by previous work [60, 54]. An additional observation we made was that non-scalable live shapes appear consistently during the runs of some benchmarks (mtrt and xalan), but only occasionally in the run of others. These latter applications exhibit life cycle dependent patterns. For example, in javac and bloat, the depth of the object-graph increases consistently throughout the run (or phase in javac) while in pmd the object graph is very deep at the beginning of the run and is consistently shallow afterwards. The SPECjbb2005 benchmark consistently demonstrated an object-graph of depth 24, for all heap sizes and in all garbage collection cycles throughout the execution. Thus, SPECjbb2005 produces a perfectly scalable object-graph and we have not studied it further in this work.

These initial results mean that the scalability of the tracing procedure with benchmarks that only occasionally manifest non-scalable object-graph shapes is sensitive to the time at which the collections are triggered. An unlucky triggering, at a time when the object-graph is non-scalable, will create a non-scalable trace and behave badly on a highly parallel platform. On the other hand, if the triggering is always lucky, i.e. it always occurs at the times when the object-graph shape is shallow, then no scalability problem arises. The probability that an execution will hit a bad triggering point depends on the length of time in which the heap is non-scalable during the run, and on the frequency of performing a garbage collection. The latter is determined by the ratio between the size of the live space and the maximal size of the heap. When the heap is made large, collections become infrequent, and for some benchmarks, this means that they are less likely to hit a point in which the live objects graph is deep enough to hurt the tracing scalability.

We proceeded by computing live objects graph depths of benchmarks at regular runs during which the object-graph was scanned at times dictated by the dynamically triggered garbage collector, but with varying heap sizes. As expected, some benchmarks (mtrt and raytrace of SPECjvm) manifested deep object-graphs during regular scans for all (sane) heap sizes while the manifestation of deep object-graph shapes with other benchmarks (mostly javac) were sensitive to the maximal heap size provided to the JVM for the run. The smaller the heap size, the more often the garbage collection is run and the greater the chance for a tracing collection to occur when the object-graph is the deepest. Different GC cycles during the run experienced
Table 2.3: Number of GC cycles and maximal and average object-graph depths for the Java benchmarks; the second column shows the heap sizes used to obtain the results.

different object-graph depths. It is possible to choose initial heap size for the run in a way that will prevent system triggered collection cycle from happening at all. As we aim to optimize GC scalability, we are not interested in runs that are not experiencing GC at all. For our experiments, we have chosen heap sizes for the benchmark in a way that will allow the benchmark to run to completion without exhausting the memory. In Table 2.3 we present the maximal and the average object-graph depths for SPECjvm and Dacapo benchmarks together with the heap sizes used to obtain these results. We also report the number of GC cycles triggered by Jikes RVM on the selected heap size. All the results that follow were obtained when running with heap sizes presented in Table 2.3. To reduce the amount of presented data, we omit data for check, compress and mpegaudio from all the tables and figures that will follow. These benchmarks are very small in terms of their heap usage and their object-graphs are typically shallow. Data for benchmarks with very deep live objects graphs that will be discussed further in this paper is emphasized in bold typesetting.
2.2 Object-Graph Shape

The object-graph depth property alone cannot be considered a sufficient indicator of how well the object-graph yields itself to a parallel trace. Consider, for example, a trace executed on $P$ parallel processors in which the object-graph consists of $P$ very long linked lists of the same length. Although the above object-graph is very deep, the potential parallelization is excellent for this number of tracing threads. We, therefore, proceeded and further investigated the shape of the live object-graphs.

We have further modified the mark and sweep stop-the-world garbage collector of Jikes RVM to record the amount of objects first discovered at each distance from roots. This gave us the information about how reachable objects are distributed on different distances from roots or on different possible depths, starting from depth 0 (roots) and ending with the object-graph depth.

We have noted that for shallow object graphs, the distribution is rather uniform while for deep object graphs, the following pattern occurs: several levels with more or less the same amount of objects at the beginning, followed by a long tail consisting of multitude of levels with only a few objects in each. Usually, the amount of live objects scanned during the cycle is very large, so the gap between the amount of objects on the initial levels and the amount of objects on tail levels is huge. This is an example of so called long tail distribution. As expected with this kind of distribution, we have found that regular statistical tools like average or standard deviation do not help to capture the distribution properties. As we have found no reliable mathematical way to meaningfully reflect object graph scalability by assigning a single number to each such objects distribution, we have turned to devising simulation based measure that depends on the number of processors available during the trace. We describe the measure in the following subsection.

For completeness, we conclude this subsection by presenting some of the objects distribution data in Table 2.4 and Figure 2.1. For each benchmark, we have chosen GC cycle with the deepest object graph and analyzed the distribution based on this cycle data. In table 2.4, we show the minimal, the maximal and the average amounts of objects on the possible levels. As we can see, for benchmarks with deep object graphs the ratio between the average and the maximal amount objects at level is greater than for benchmarks with shallow object graphs and the skewness of a distribution is larger.

In Figure 2.1, we show the distribution of live object depths. For each possible depth (on the x-axis), the value on the y-axis depicts how many objects with this depth exist in the object-graph. Note that for all the benchmarks, the number of objects (on the y-axis) is shown on a logarithmic scale. Furthermore, for benchmarks with deep object-graphs, we had to put the depths (on the x-axis) on a logarithmic scale as well, to make the data in the graph visible. These graphs show the long-tail distribution of object depths for javac, mtrt, bloat, pmd and xalan benchmarks. For
Figure 2.1: Object distribution among the different depths. The x-axis represents the depth and the y-axis represents the number of objects found at that depth.
Table 2.4: The summary of live objects distribution: total amount live objects, live object graph depth, maximal and average amounts of objects on different distances from roots (depths).

<table>
<thead>
<tr>
<th>Name</th>
<th>Depth</th>
<th>Total</th>
<th>Max</th>
<th>Avg</th>
<th>Skewness</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>16</td>
<td>385,168</td>
<td>136,239</td>
<td>24,098</td>
<td>2.2</td>
</tr>
<tr>
<td>jack</td>
<td>38</td>
<td>108,881</td>
<td>33,312</td>
<td>2,884</td>
<td>3.2</td>
</tr>
<tr>
<td>javac</td>
<td>1,234</td>
<td>383,223</td>
<td>41,148</td>
<td>312</td>
<td>10.6</td>
</tr>
<tr>
<td>jess</td>
<td>32</td>
<td>124,089</td>
<td>33,209</td>
<td>3,928</td>
<td>2.6</td>
</tr>
<tr>
<td>mtrt</td>
<td>1,416</td>
<td>455,608</td>
<td>67,894</td>
<td>322</td>
<td>14.4</td>
</tr>
<tr>
<td>antlr</td>
<td>60</td>
<td>230,313</td>
<td>53,299</td>
<td>3,875</td>
<td>2.9</td>
</tr>
<tr>
<td>bloat</td>
<td>1,195</td>
<td>378,555</td>
<td>72,857</td>
<td>319</td>
<td>15.2</td>
</tr>
<tr>
<td>hsqldb</td>
<td>47</td>
<td>3,360,562</td>
<td>598,017</td>
<td>71560</td>
<td>2.7</td>
</tr>
<tr>
<td>jython</td>
<td>128</td>
<td>466,254</td>
<td>80,363</td>
<td>3,804</td>
<td>4.2</td>
</tr>
<tr>
<td>lusearch</td>
<td>38</td>
<td>215,039</td>
<td>51,594</td>
<td>5,727</td>
<td>2.5</td>
</tr>
<tr>
<td>pmr</td>
<td>18,482</td>
<td>434,302</td>
<td>51,942</td>
<td>23</td>
<td>57.1</td>
</tr>
<tr>
<td>xalan</td>
<td>8,476</td>
<td>371,966</td>
<td>52,607</td>
<td>44</td>
<td>34.2</td>
</tr>
</tbody>
</table>

example, in the heap built by xalan, the majority of the objects have depths of 17 or less. After depth 18 the shape of the object-graph becomes more and more narrow, until, starting from depth 39 there are only two objects at each depth.

In the remainder of this work, we will denote by heap shape the shape of the live object-graph in the heap.

2.3 Idealized Trace Utilization Measure

In Section 2.1 we have started the investigation of the influence of the object graph properties on the trace scalability by measuring and analyzing the object graph depth. We saw that while depth measurements provide some information, this information is not directly related to the parallelizability of the benchmark. Heap shape visualization presented in Section 2.2 helps to grasp the problematic properties of heap shapes, but a more precise and convenient tool is needed to assess and analyze these properties. This need led us to devise the idealized trace utilization measure described next and used extensively thereafter.

Our goal is to capture and quantify the inherent heap shape properties relevant to the scalability of object graph trace and independent of other factors such as the particular trace implementation and the load balancing scheme. We choose not to
directly measure the tracing time when running with different amount of tracers for this would bear the influence of a particular JVM implementation we are using and a platform we are running our tests on. Instead, we consider and simulate a simplified and clean version of a trace run with a given number of processors. Recall that garbage collection tracing threads have to maintain a list of objects to be scanned. Tracing work consists of iteratively picking one object from the list, marking it, and inserting its unmarked descendants into the list. The tracing algorithm needs to take care of coordinating the tracing threads, in particular, the process of selecting an object to be scanned and the process of inserting objects into the list. In order to get a well balanced work distribution, the algorithm needs to make sure that the work is evenly split. To distill the scalability properties of a given heap shape, we imagine a clean and idealized tracing procedure that ignores all these and other implementation and platform dependent issues. We make the following idealizing assumptions:

1. Perfect load balancing. Namely, the list of objects that have been found reachable but not yet traced is accessible by all the tracing threads with no load-distribution or synchronization problems.

2. Any object is scanned in a single clock tick\(^2\). That is, within a single clock tick, a tracer thread can atomically pick an object \(A\) from the work list, mark \(A\), find all objects directly reachable from \(A\), discover which of them are not marked, and mark and add these to the list.

3. Ideal memory subsystem behavior - no latencies due to cache misses, conflicts or false sharing.

4. BFS style of traversal which is geared towards higher parallelism.

These assumptions imply that \(P\) tracing threads on \(P\) processors can trace \(P\) objects from the list (and add their descendants to the list) in a single clock tick. We call such trace runs idealized traces. The only remaining difference between such clean runs on different heap shapes of the same size is whether the work list contains \(P\) elements to be handled in each clock tick. If not, some processors become idle, and scalability is hindered.

For example, if the object-graph is a single linked list, then at any point in time the list of objects to scan consists of a single object and all tracing threads, except for one, remain idle throughout the trace. This happens because, at each clock tick, the single available object is pulled (by one of the threads) from the list, but scanning it only yields a single object that is added to the list instead of the one that was just pulled out. Therefore, at all clock ticks there are \(P - 1\) idle tracing threads.

\(^2\)Again, we use the term clock tick to denote the time it takes to execute a single computation step. In our idealized procedure, we assume synchronized cores, and ignore cache misses, page faults, interrupts, etc.
The fraction of idle time during the trace is $1 - \frac{1}{P}$ and this fraction is monotonically increasing with $P$.

Informally, we base our tracing scalability measure on knowing how many objects are available in the list at each clock tick. Of course, this number depends on the number of tracing threads (denoted by $P$), because at any clock tick $t$, each processor pools an object from the work list so the total of $P$ objects are pulled and scanned for referents. By assuming a BFS-style traversal, which is geared towards higher parallelism, we are able to demonstrate non-scalability even with respect to the most parallel-friendly tracing procedure. Note also that the resulting measure is inherent to the heap shape. It is independent of both the collector implementation and the particular hardware employed. Also, it can be computed on a single processor, i.e., evaluating this measure does not require a many-core platform.

We have further instrumented our single threaded breadth first search trace in order to dynamically compute empirical estimation of how scalable the tracing would be if done by multiple tracers. To guide and structure the computation, we need several definitions as follows:

- Denote the amount of processors available for the trace phase by $P$.
- Denote the amount of objects ready to be scanned at time slot $t$ during the $P$-processor trace by $A_t(P)$. Under our assumptions, this is the amount of work units available for tracers. If at some moment of time, $t$, amount of available work, $A_t(P)$, is greater than the number of tracers, $P$, every tracer has work to do and no tracer stalls. If $A_t(P)$ is less than $P$, not every tracer has work to do and some tracers stall.
- Note that at time 0, objects ready to be scanned are the root objects and their amount is known. At every other time $t$, objects ready to be scanned are objects that were not scanned at time $t - 1$ and objects that were discovered at time $t - 1$.
- Denote the amount of new objects discovered by $P$ tracers at time slot $t$ by $N_t(P)$.
- Consider processor slots spend to process objects available for trace to be *useful work* and denote the amount of useful work done by $P$ tracers at time slot $t$ by $U_t(P)$.
- Consider the idle processor slots to be *wasted work* and denote the amount of work wasted by $P$ tracers at time slot $t$ by $W_t(P)$.

Note that the following simple dependency holds:

\[
\text{if } A_t(P) < P \text{ then }
\]
\[ U_t(P) = A_t(P) \]
\[ W_t(P) = P - A_t(P) \]

else
\[ U_t(P) = P \]
\[ W_t(P) = 0 \]
end if

We have used our modified single threaded BFS heap trace and added runtime counters to keep track of \( A_t(P), U_t(P), W_t(P) \) and the number of time slots required to complete the trace. We maintain these counters for several possible numbers of tracers simultaneously - from 1 to 1024 in powers of 2. We initialize the time slot counting with zero and \( A_0 \) with the amount of roots. We then simulate time slot \( t \) by letting \( U_t \) objects to be scanned for references and, as a result, new \( N_t \) objects to be discovered. Knowing \( N_t \), we compute the amount of objects available for scan at the next time slot as \( A_{t+1} = A_t - U_t + N_t \). Following this process stepwise we can compute \( A_t, U_t, \) and \( W_t \) for every moment from the beginning till the end of trace, counting the amount of simulated time slots required to complete the process. Tracing is completed when \( A_t \) is zero. We use the following notation for the accumulated values of the useful and wasted processors work and the tracing time:

- **Total Time** \( (T(P)) \) is the amount of time slots required by \( P \) tracers to complete the trace.

- **Total Useful Work** \( (U(P)) \) is the total amount of useful work done by \( P \) tracers during the trace. Can be either computed as a sum of \( U_t(P) \) for every \( t \) or observed as being equal for the total amount of objects scanned during the trace.

- **Total Wasted Work** \( (W(P)) \) is the total amount of work wasted by all the tracers during the trace. Computed as a sum of \( W_t(P) \) for every \( t \).

At the end of our modified BFS trace, knowing the values of \( U(P), W(P) \) and \( T(P) \), we compute the following useful metrics:

**Processor Utilization** is a percentage of useful work in all the work done by \( P \) tracers; computed by

\[
Util(P) = 100 \cdot \frac{U}{U + W}
\]

**Stalls Percentage** is a percentage of wasted work in all the work done by \( P \) tracers; computed by

\[
Stalls(P) = 100 \cdot \frac{W}{U + W}
\]
Figure 2.2: Worst case idealized trace utilization for various levels of parallelism. The lines of *javac* and *bloat* are hard to distinguish as they almost collide, due to similar behavior.

**Speedup** is a ratio of amount of time slots required to complete the trace by a single processor to the amount of time slots required by *P* processors; computed by

\[
\text{Speedup}(P) = \frac{T(1)}{T(P)} = \frac{U}{\text{Time}(P)}
\]

Note that all three values above - speedup, utilization and stalls percentage are equivalent and can be used interchangeably to quantify the scalability of the object-graph being traced. Whichever of three is to be used is a matter of choice and convenience. In what follows we’ll use utilization as our primary object-graph scalability measure and call it the *idealized trace utilization* measure.

We acknowledge that our computation has no way to account for the different possible orders in which objects available for tracing can be stored. Different orderings will lead to choosing different objects to be scanned in the next time slot and so will result in different values of \(N_t\) and in different set of objects available to be traced at the next moment. Still, the resulting measure is useful for capturing important scalability related object-graph properties.
2.4 Idealized Trace Utilization Measurements

We ran a modified version of Jikes RVM, computing the idealized trace utilization, as the heap shape scalability measure. Our measurements cover the SPECjvm98 and the DaCapo benchmark suites, as it was established earlier that SPECjbb2005 does not manifest deep heap shapes. We computed the utilization measure for each GC cycle and then observed the average and minimal values (over all the collection cycles) for each benchmark run. Results are presented in Figures 2.2 and 2.3, where we show the worst case and the average idealized trace utilization as a function of the number of working processors $P$ for every benchmark.

Recall that idealized trace utilization measure is geared towards demonstrating non-scalability, as it runs imaginary perfectly-coordinating parallel tracing threads. When this measure shows bad utilization, we know that scalability is a problem. When it shows good scalability, it is not clear whether such good load distribution and speedups are attainable on a real system.

It can be seen that for up to eight processors, typical of many of today’s parallel platforms, the average-case scalability of all the benchmarks looks good. Even on the worst-case measurements, benchmarks exhibit up to 15% idle time, which is still reasonable. For 32 processors, a level of parallelism available today, we start to see substantial idle times, which naturally increase when the level of parallelism goes up. To diminish the clutter in our graphs, we do not show data for benchmarks with worst case utilization values above 90%. Out of the fifteen benchmarks measured above,
five came out as problematic for tracing on highly parallel platforms: javac, bloat, mtrt, pmd and xalan.
Chapter 3

Improving the Object-Graph Scalability

Measurements presented in Section 2.4 reveal the existence of Java benchmarks that generate heaps with inherently non-scalable shapes and can limit the scalability of parallel tracing. Poorly-scalable heap shapes should be avoided, since they may prevent Java applications from taking advantage of multiple computation cores. In some cases application writers can avoid creating such structures. There are cases, however, where data structures grow dynamically during a long application life time according to data or events coming from external sources. In such cases, runtime systems should be able to handle the resulting structures and to avoid problems that can result from certain patterns in them. There are many possibilities for such handling and we can think of interesting directions involving collection classes design, compiler aided optimizations etc.

In this work we investigated two solutions where problematic object graph patterns are handled by the garbage collector. In the following subsections we describe these solutions, in Section 4 we provide the implementation details of our prototypes, and in Section 5 we describe the evaluation methodology and present and analyze the results.

3.1 Adding Shortcut References

The first approach we investigated aims at modifying the object-graph structure by adding new references that are invisible to the application, but useful for the tracing threads. Intuitively, this can be seen as an attempt to ensure that at each moment of the trace there are enough objects available to be scanned and these objects have enough unscanned references in them so enough work is produced to keep all the tracers busy at the next time slot, etc. In terms of the object graph shape discussed
in Section 2.2 this can be seen as trying to achieve a more even distribution of live objects among different distances from tracing roots and to reduce the maximal heap depth. The goal is to decrease the depth of live heap objects, by creating shortcuts into the deepest parts of the long and narrow data structures. This yields a shallower graph, which in turn, makes more work available for the tracers to execute at earlier stages of the trace.

A related theoretical optimization problem is to find the minimum set of edges that would improve the parallel trace efficiency by a given threshold. This approach can be formulated as a problem of having initial object graph with certain scalability properties and asking a question how the scalability properties can be improved by introducing as few as possible additional edges among objects in the graph. In this case, we are not changing the root set that is used to initialize the tracing (and the scalability measure computation). This problem is interesting in itself as a graph theoretical exercise.

However, in this work we focused on practical solutions and their plausibility and efficacy. Every practical implementation of such a scheme requires answers to the following set of questions:

- How many references we allow to be added to a single object? From a practical standpoint, adding a reference to an object means allocating memory to hold the new reference. Answering this question involves memory related trade off and is directly related to another important question, namely, how many objects we allow to be modified.

- Where to introduce the new references? This question involves choosing both the source and the target of the newly introduced reference. It is possible to use the dynamic data collection schemes, possibly aided by data collected at compilation time and at object allocation time. Relevant class or allocation site information is whether the object is likely to be part of the recursive data structure.

- When to introduce the new references? Again, there are a lot of possibilities here. New references can be introduced in Java code by the implementers of the standard collections’ classes, by the compiler using runtime hints, by some specialized runtime helper threads or by GC threads on attempt to improve the next cycle’s tracing scalability.

- How to introduce the new references? The simplest way is to allocate space to hold the additional references for every heap object. This space can be allocated either in the object header or in a side table. The disadvantage of such a scheme is the unnecessary memory overhead, because not every heap object will need new references to be added. More advanced schemes can use runtime generated
data to trigger dynamic class rewriting and recompilation so that the resulting object structure will have more references slots specifically where these slots are needed.

- How to maintain the new references? Namely, should additional references remain in place or be removed? Here we note that no correctness problems can be introduced by leaving such references in place. On the other hand, new references can keep alive objects unreachable by the regular references thus creating a floating garbage\(^1\) if not timely removed. In addition, the object graph evolves during program life time and some of the additional references may become unnecessary and just generate unneeded work for the garbage collector and take up space.

Answering the above questions differently leads to a large space of possible solutions. We have prototyped one of the possible solutions along with the method for assessing the potential of the shortcuts approach. Our prototype introduces one additional reference slot in the header of each object in the heap, initiating it with Null. During the benchmark run, each time the program is paused for garbage collection, the prototype performs its heap shape improving work as follows: first, normal garbage collection trace, modified to compute the idealized trace utilization of the heap, is run; second, the algorithm computing new references sources and targets is run as described below and, as a result, shortcuts are added to the heap; third, the trace computing the idealized trace utilization is run again to evaluate the effect; then the cycle proceeds as usual to completion. The prototype removes all the shortcut references after each cycle so they have no bearing on the next cycle (this happens during the first regular trace of the next cycle). The shortcut adding algorithm can be run by several tracers simultaneously with no additional synchronization. Note again that this scheme cannot fail the correctness of the garbage collector as no object can cease to be reachable as a result of the process.

We next describe the shortcut adding algorithm or how shortcuts are chosen to be added to the heap. Jikes RVM prototype implementation is presented in Section 4.2.5. To evaluate our prototype, we computed the idealized trace utilization measure, described in Section 2 above, before and after the shortcuts were added. Measurements and results are presented in Section 5.1.

### 3.1.1 Shortcut Adding Algorithm

Consider a heap object \(O\) and its reachability subgraph, i.e. all objects reachable from \(O\) in the heap. We need to decide whether a shortcut should be added to \(O\), and if it

\(^1\)Floating garbage includes all unreachable objects that the collector does not identify as such, and does not reclaim.
Figure 3.1: An example of a linked list with shortcuts added. Here, the shortcut length is 4 and the shortcut distance is 2.

should, we need to select an object in $O$’s subgraph as the target. To decide whether a potential shortcut is useful, we categorize the subgraph of $O$ by two parameters: the size of the subgraph, i.e. how many objects are reachable from $O$, and the depth of the subgraph, i.e. the distance of $O$ from the node that is farthest from it in the subgraph. We only install a shortcut in $O$ if the size of its subgraph is larger than a predetermined threshold $\text{size}$, and if the ratio of the depth to the size of the subgraph is larger than a predetermined threshold $\text{ratio}$. Note that the largest possible ratio value, which is 1, is obtained for a linked list structure.

Once a candidate object is found with a subgraph of appropriate size and depth-to-size ratio, we add a single shortcut to it. It does not make sense to make the deepest object in the graph the target of the shortcut, because letting the trace jump to the end of the structure is not helpful. We, therefore, set a parameter to define the length of the shortest path between the shortcut source and the shortcut target in the subgraph. We call this parameter $\text{shortcut length}$; its value determines the length of all the shortcuts added by the algorithm.

Often, when an object is a candidate for a shortcut installation, then its parent is also a good candidate. This is clearly the case for a linked list. However, it is not very effective to install shortcuts both in an object and in its parent, leading to a target and the target’s parent. Thus, we set an additional parameter to define a distance between the two shortcut sources along the same path. This parameter is denoted $\text{shortcut distance}$. An example of a list with shortcuts of shortcut length of 4 and shortcut distance of 2 is depicted in Figure 3.1.

Overall, the algorithm has four parameters: the minimal candidate subgraph size ($\text{size}$), the minimal candidate subgraph depth-to-size ratio ($\text{ratio}$), the shortcut length and the shortcut distance. The algorithm is a modified DFS traversal that upon retreating to object $O$ during the traversal, retains enough information about $O$’s subgraph to be able to evaluate its size and depth as well as addresses of the nodes along its deepest branch. Knowing that, the algorithm decides whether shortcuts should be added in $O$’s subgraph. If yes, the algorithm has enough information to compute a target object in the subgraph whose depth is shortcut-length with respect
to $O$ and an additional candidate shortcut source object whose depth is \textit{shortcut-distance} with respect to $O$. At this point, the tracer can install a shortcut from $O$ to the target and, possibly, other shortcuts along the deepest path in $O$'s subgraph.

In order to describe the algorithm, we make one additional observation of the DFS traversal and its way to keep the already discovered but not yet scanned objects in a stack. When object $O$ is first discovered, its address is added to the stack. At that time the stack can contain other objects that the traversal needs to keep track of. We denote the amount of the objects in the stack at the moment $O$ is pushed onto it as $O$'s \textit{level}. The name comes from seeing the stack as a vertical addresses array with the fixed bottom and the top that can go up and down as items are pushed and popped thus changing the top item’s level.

The algorithm uses auxiliary data structure called the Branch History Table (BHT) that holds references to objects on the longest branch of the currently scanned subtree. References are added to BHT when they are popped off the DFS tracing stack and scanned for referents. For each added object reference, object’s level in the DFS stack and the amount of its direct referents is held. After the object reference is added to BHT and as long as its successors are being scanned, as detected by comparing the level of the objects in the DFS tracing stack, new references are added to BHT. As soon as a reference popped off the DFS tracing stack is recognized to be out of the most specific currently scanned subtree (again, as detected by comparing the level of the objects in the DFS tracing stack), the most specific subtree that was just fully scanned is processed as follows:

1. Data on the amount of direct referents of the objects in a subtree is accumulated starting from the subtree bottom to its top thus producing data on the amount of objects in a subtree and its depth.

2. Subtree size and depth is judged according to the values of \textit{size} and \textit{ratio} parameters and it is decided whether there is a need to introduce shortcuts in the subtree.

3. If needed, shortcut references are introduced into the objects along the longest path in the subtree and according to the values of \textit{shortcut length} and \textit{shortcut distance} parameters.

4. References of the subtree objects are removed from the BHT; the subtree size and depth information is accumulated into the counters of the most immediate predecessor of the subtree’s root if there is one in the table. There may be no such an object if the subtree just processed was rooted by the application root.

When the fully scanned subtree processed as described above, the newly popped object reference can be added to the BHT and the process continues.
3.1.2 Possible Optimization

When adding object references to the BHT, we inspect the number of object’s direct referents. It is clear that if the object has no referents (it is a leaf in the live object graph), there is no point to adding it to the structure as it will be removed after the next scanned reference is inspected. It is also clear that this reference cannot be a root to the deep and narrow subtree so will never trigger shortcut addition. Such references are processed by just updating the BHT entry for their most immediate ancestor if one is held in the table at the moment.

In addition, although an object having many direct referents can be a root of the very deep and narrow subtree, it must be that at least one of its referents is the root of the very deep and narrow subtree as well. Objects with a lot of references can be excluded from the BHT leading to less memory and compute overhead. It is up to the implementation to decide what amount of direct references will prevent an object from being held by the BHT.

3.1.3 Performance Considerations

Computation wise, the shortcut addition algorithm can piggy-back on otherwise required heap traversal and does not add a lot of overhead to it.

In terms of memory requirements, the algorithm requires two types of investment: the place to keep new inter-object references and the place taken up by the Branch History Table.

Overhead of the first type can be controlled by collaboration with the compiler and object allocation code to induce additional space in object headers only when needed. Other solution would be to piggy-back on existing in memory heap maps that have to be looked up during the trace process, for example the mark bits table.

BHT overhead can become annoying even if the optimization described in Section 3.1.2 is used. The table contains an entry for every object on the longest path of the most inclusive currently scanned subtree and this amount can be large when deep data structures are present. If there are no deep data structures, the overhead is small. This is a good property meaning that if the application cannot enjoy algorithm’s benefits, it will not have to pay for its execution. We will show in Section 5.1 that in our prototype this property holds indeed. To reduce memory requirements of the BHT for application with deep heaps, collapse techniques can be introduced. For example, if more than certain amount of references along a single path is introduced, the data can be accumulated for some subpaths and the entries contributing to the accumulation can be removed. This technique trades off the memory overhead for data structure complexity, i.e. computational overhead and software creation and maintenance costs. The trade off has to be decided on and engineered for every specific implementation.
3.2 Tracing Randomly in Parallel

The second approach we investigated does not modify the heap shape but allows idle processors to trace the heap starting from objects not yet known to be alive. The idea is that when a deep data structure actually exists in the heap, the trace cannot be well parallelized and processors become idle during the tracing time. Such idle threads can be used to trace objects speculatively, starting from random heap objects and, when lucky, aid the tracing effort and improve the trace scalability.

This direction can be formulated theoretically as a problem of taking the object graph with certain scalability properties and a set of roots to start the trace from and asking a question what additional heap objects can be chosen as roots to facilitate better trace scalability. In order to be helpful, such additional roots must be live heap objects far enough both from the original roots and from the deepest heap objects. In practice, however, it is not possible to know whether an object is alive before it is reached by the tracing process, when it is too late for it to be chosen as an additional root.

When allowing tracing from the objects with unknown reachability status, we must be able to distinguish objects discovered when tracing from the additional roots from objects discovered when tracing from regular roots. To accomplish this, we have introduced additional tracing colors – one color per additional tracing root and a way to keep track of objects’ liveness states in presence of these different colors: Denote the effort of the normal tracing from the program roots as the main trace. While the main trace is executing, a thread that becomes idle can select at random an object in the heap that has not yet been marked. It then traces the descendants of this object, marking the objects and its descendants with a special color uniquely assigned to this thread, say red. If the main tracing procedure discovers that a red object is reachable from the roots, then the entire red component is declared reachable and the front of the red trace is added to the work list of the main trace.

This method, though simple to describe, has some problems. First, if the main trace hits a red object, we can be sure that some of the red objects are alive, but not all of them must be alive. It is possible that the main trace discovered a reachable red object, but one that is deep in the trace of the helper thread. All the object’s predecessors in the helper’s trace may actually be unreachable. An example is depicted in Figure 3.2. So if we do not want to trace from the reachable red object and determine accurately which red objects are reachable, then we need to assume conservatively that all red objects are reachable. This creates an inaccuracy in the trace and implies floating garbage.

The most important design questions for such an approach are:

- How to choose additional tracing roots? This is the most important question because picking up ‘bad’ additional roots can render the whole approach more harmful than helpful. Choosing a dead root can cause tracing through a lot
Figure 3.2: An example of a red trace, which is hit by the main trace. Note that only some of the red vertices are reachable. The rest create floating garbage.

of dead objects which can lead to wasted efforts in the best case and a lot of floating garbage in the worst. The simplest way is to pick roots at random. The next logical step is to filter randomly picked objects for some desired properties, like the number of referents, objects size or object type. Other possibility would be to use more specific hints collected by runtime helpers when the program is run – like which objects were created or updated recently and have higher probability to be live or which objects have higher probability to be part of a recursive data structure.

• When to start tracing through the additional roots? Possible choices are: to start additional tracing between the GC cycles, at the beginning of the cycle or at some point during the regular trace, e.g. when uneven load balancing is experienced.

• What resources to use for tracing through additional roots? It is possible either to use additional helper threads that run on idle processors or to use regular tracer threads that become idle or succeed to achieve only very small work packets.

As with the shortcuts approach, we have prototyped one specific solution for this approach as well. Our prototype uses helper tracer threads that pick up random roots. For each root the helper obtains a color and traces from the chosen root using the chosen color in a breadth first search manner. Helpers are spawned together with the regular tracers just after the roots are collected. The algorithm is conservative so that it never considers live object dead but does in some cases considers dead objects live. To restrict the amount of floating garbage, we limit the number of objects that a helper thread marks with any single color by the trace-limit parameter.

The algorithm consists of three major components - the modified regular trace routine, the helper trace routine and the completion phase. In our implementation, the regular tracers and the helper tracers work together all the time starting with the point where roots are available and till regular tracers exhaust their stacks.
this point, the completion phase of the algorithm is executed. Upon completion, the algorithm decides for every heap object whether it has survived the collection cycle. This decision must propagate to the sweep phase so that the allocator knows what objects can be recycled. Next, we describe the algorithm’s auxiliary data structures and major routines. In section 4.2.6 we present our Jikes RVM prototype implementation in more detail and in Section 5.2 we report and analyze the evaluation results.

### 3.2.1 Colors Table

Colors table is a simple data structure that keeps one entry per additional tracing color. For every color, it contains:

- The root reference that has started this color’s trace.
- The liveness status that initially set to false and is changed to true by the main tracer that encounters object colored in this color.
- A list of other additional colors that are encountered by the tracer using this color.
- A list of objects that are colored by this color but were not scanned because the trace-limit was reached. Tracing from these objects will have to be resumed if the color will be determined to be live.
- Flags required to synchronize the trace completion.

### 3.2.2 Main Trace

Threads performing the main trace must take into account that helper threads trace through the heap marking objects they discover in custom colors. When regular tracer encounters object marked by a custom color, the regular tracer does not continue to trace through this object and marks the color as ‘live’. Later on, all the objects marked by ‘live’ colors will be conservatively considered live and will survive the collection. Then the regular tracers exhaust their work lists, they start the additional trace completion phase required to trace from all the unscanned objects colored by the live colors.

### 3.2.3 Helper Trace

Threads performing the helper trace work in a loop consisting of choosing a root, obtaining a custom color and tracing objects starting from this root till completion or till the trace-limit of objects are traced, whichever comes first. For every object the helper encounters during this process, it checks whether the object was already
marked. If the object was marked by the main trace or by the helper’s own custom color, the helper does not continue tracing though it. If the object was marked by custom color other than its own, the helper adds the color to its list of ‘depending’ colors and does not continue tracing though the object. If the object was not marked yet, the helper proceeds as usual – marks the object by its custom color and continues tracing though the object. If the helper’s trace ends because the discovered objects list is exhausted, no further action is required and the helper proceeds with the next loop iteration - picking new root, new custom color, etc. If the helper’s trace ends because the limit is reached, there still can be object references on its tracing stack. In this case, the helper saves these discovered but not yet scanned references before proceeding to the next iteration.

### 3.2.4 Trace Completion

To complete the trace process, it is required to compute the transitive closure on the color ‘dependency’ relation to decide which colors are transitively reachable from the main trace color and which are not. Colors reachable from the main trace color are ‘alive’ while other colors are dead. For every live color that has stored discovered but not yet scanned references, it is required to complete tracing from these saved references. Note that during the process, the trace can encounter objects colored by other colors, thus turning more colors live and, potentially, requiring their stored objects to be traced as well. The trace is completed when where are no live colors with untraced saved references left.

### 3.2.5 Random choices with Filters and Biases

In the simple scheme, unmarked objects are chosen randomly in the heap, and their descendants are traced. We note again that if a dead object is chosen, then its descendants are traced in vain, and effort is wasted. Therefore, the question arises whether we can bias the choice of objects to be more effective.

Indeed, our initial experience with the method was not very encouraging because a lot of the idle threads’ work was in vain. The main problem is that the chances of picking a dead object and tracing its descendants are high to start with, and they monotonically increase as the live objects become marked. This problem is reflected in our initial experiments. Can this method be improved?

In general, it is possible to further filter randomly picked objects for some desired properties, like the number of referents, objects size or object type. Another possible solution would be to use more specific hints collected by runtime helpers when the program is run, such as information about objects that were recently updated and thus have higher probability of being alive, or compile-time information on which objects have a higher probability of being a part of a recursive data structure.
One good bias that is obtained for free in our Jikes RVM based implementation is that it only considers blocks that have been allocated or that are ready for allocation. It does not consider large, free spaces that are not yet allocated. Nonetheless, this was not enough to make the method a winner. We, therefore, made an ad-hoc check to see whether additional information can help. In particular, we added the following test for every inspected object. We accessed the object’s type information at runtime, and checked whether the object had a reference to an object of the same type as its own. On the positive side, this test improves the likelihood that a linked list will be chosen. On the other hand, this test does not increase the probability of choosing live object and is only a prototype meant to validate the potential of further optimizations.

### 3.2.6 Performance Considerations

In terms of memory requirements, the algorithm needs additional mark bits per object to increase the amount of different mark colors. In addition, the color table needs to be stored in memory. The memory overhead can be controlled by the additional parameters specific to each implementation.

Computational overhead of this algorithm can be a problem, though. In addition to the inherent algorithmic problems of performing the unnecessary work and creating floating garbage, the implementation requires careful synchronization between the collector threads. Synchronization is required both during the tracing phase, to synchronize the access to the object’s color marks, and during the trace completion phase. Recursive nature of the trace completion phase makes it vulnerable to the engineering errors. Provably correct completion phase can turn to be rather long. As the whole purpose of the approach is to reduce the tracing phase time when multiple processors are available, long synchronized trace completion phase can undermine the method’s intended benefit.
Chapter 4

Implementation

4.1 Instruments and Equipment

4.1.1 The Hardware and OS Platform

The hardware platform used for this research is IBM x3400 system. The computer features 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors so the maximal amount of parallelism that can be achieved on it is 8-way. The operating system is Red Hat Enterprise Linux Server release 5 (Tikanga). To explore the scalability of existing benchmarks and to evaluate the proposed solutions in terms of the time required to trace the heap, a computer with more cores will be required.

4.1.2 The Language Platform

The language platform used for this research is Jikes RVM version 3.1.0 [2]. Jikes RVM is an open-source research virtual machine for Java originating from IBM Research project called Jalopenio [6] and written almost entirely in Java. Today Jikes RVM includes implementations of cutting-edge Java virtual machine technology making it a popular research vehicle. There are now over 188 publications and 36 dissertations listed on the Jikes RVM publications site.

Two key advantages of using Jikes RVM as a platform for this research are the source code level availability and the existence of the Memory Management Toolkit (MMTK) – an efficient, extensible, and portable framework for building garbage collectors [16]. On the negative side, as almost all of the Jikes RVM is written in Java, the runtime VM components employ Java objects that are subject to garbage collection along with the Java application objects. This includes objects created and used by the garbage collector itself. Studies involving Java objects demographics can be biased by inclusion of the non-application objects if care is not taken to distinguish the VM objects from the application objects. In addition, there are some
minor technical inconveniences of implementing GC in Jikes RVM: the fact that in
garbage collector code no dynamic memory allocations are allowed so that all the
memory the garbage collector uses must be allocated before the cycle is started; and
the need to use ‘magic’ methods to reference to the non-object memory locations [33].

The fact that much of Jikes RVM is written in Java means that with Jikes RVM,
every benchmark is larger than usual: executing even the simplest ‘Hello world’ pro-
gram requires executing Jikes RVM itself. Jikes RVM, depending on configuration,
can consist of about 1,000 classes with a total class file size of about 5,000 KB. In
addition, both Jikes RVM and the application require standard Java libraries, which,
depending on the version, add up to about 2,000 classes with a total class file size
of more than 3,000 KB. Thus, the Jikes and the standard library classes have much
more Java code than, for example, the code of the benchmark javac (1,909 KB),
which is one of the largest benchmarks in common use in Java memory management
research. The size of the Java code of Jikes RVM and the libraries impacts the re-
sults of experiments with Jikes RVM and make them different from the results with
industrial JVMs implemented in lower level languages.

A Jikes RVM boot image is configurable and can include one of several garbage
collectors implemented as part of MMTk. One of the most basic collectors is a mark
sweep stop the world collector. We have implemented new garbage collector flavors in
Jikes RVM, extending the parallel mark and sweep stop the world collector of MMTk
and relying on some support from the existing generic MMTk code base. Properties
of this and other parallel collectors implemented in MMTk are studied in [9].

4.1.3 The Benchmarks

We have used standard Java benchmark suites - SpecJVM98, SPECjbb2005 [4] and
DaCapo [1]. Table 4.1 presents the benchmarks’ memory usage properties: the size
of the code, the amount of allocations and the maximal size of the live objects set,
as reported in the research literature [24], [15]. Then the benchmarks are described
in more details relevant to the current work.

SPECjvm98 consists of eight programs: check validates the correctness of the VM
and the other seven are used for computing the performance score. For all the pro-
grams, SPEC provides three different inputs referred to as ‘problem size 100, 10, and
1’. All our runs use the largest input size. Some of the programs (compress, jack,
javac) iterate multiple times over the same input; all programs except mtrt are single-
threaded. Today, SPECjvm98 programs are too small compared to the modern Java
applications and the suite is considered outdated. System Performance Evaluation
Corporation (SPEC) has published a new suite, called SPECjvm2008 that is com-
piled of a larger set of more realistic and complex programs. Unfortunately, the Jikes
RVM version we have started to work with (3.0.1) is not able to run the SPECjvm2008
suite because of the incomplete support of the required console functionality. This
<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>code size</th>
<th>max live heap</th>
<th>allocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>Simple JVM correctness check</td>
<td>5.8</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>Modified Lempel-Ziv method</td>
<td>17.4</td>
<td>6.7</td>
<td>105</td>
</tr>
<tr>
<td>db</td>
<td>Series of transactions on a small database</td>
<td>9.9</td>
<td>7.2</td>
<td>231</td>
</tr>
<tr>
<td>jack</td>
<td>Java parser generator</td>
<td>129.4</td>
<td>1.2</td>
<td>61</td>
</tr>
<tr>
<td>javac</td>
<td>Java compiler run 3 times</td>
<td>548.3</td>
<td>6.5</td>
<td>111</td>
</tr>
<tr>
<td>jess</td>
<td>Expert system based on NASA’s CLIPS</td>
<td>387.2</td>
<td>1.1</td>
<td>161</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>ISO MPEG Layer-3 audio decoder</td>
<td>117.4</td>
<td>very small</td>
<td></td>
</tr>
<tr>
<td>mtrt</td>
<td>3D raytracer rendering a scene</td>
<td>56.5</td>
<td>3.0</td>
<td>147</td>
</tr>
<tr>
<td>antlr</td>
<td>Parses grammar files</td>
<td>212.7</td>
<td>1.0</td>
<td>237.9</td>
</tr>
<tr>
<td>bloat</td>
<td>Performs analysis on Java bytecode files</td>
<td>169.1</td>
<td>6.2</td>
<td>1,222.5</td>
</tr>
<tr>
<td>hsqldb</td>
<td>Executes a JDBCbench-like benchmark</td>
<td>130.2</td>
<td>72.0</td>
<td>142.7</td>
</tr>
<tr>
<td>jython</td>
<td>Interprets the pybench Python benchmark</td>
<td>462.5</td>
<td>0.1</td>
<td>1,183.4</td>
</tr>
<tr>
<td>lusearch</td>
<td>Uses lucene to do a text search of keywords</td>
<td>65.5</td>
<td>10.9</td>
<td>1,780.8</td>
</tr>
<tr>
<td>pmd</td>
<td>Analyzes a set of Java classes for problems</td>
<td>152.4</td>
<td>13.7</td>
<td>779.7</td>
</tr>
<tr>
<td>xalan</td>
<td>Transforms XML documents into HTML</td>
<td>126.2</td>
<td>25.5</td>
<td>60,235.6</td>
</tr>
<tr>
<td>SPECjbb2005</td>
<td>Three-tier client/server system</td>
<td>69.7</td>
<td>varies (configuration)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Properties of benchmarks: the code size in KBytes, the maximal live heap size in MBytes, and the total allocation size in MBytes.
is the reason we have used the old, but still widely used by the research community, SPECjvm98 suite. Here is the list of SPECjvm98 programs with the short description of each:

**compress** implements file compression and decompression using the modified Lempel-Ziv method. It performs five iterations over a set of five archive files, each of them between 0.9 Mbytes and 3 Mbytes large. Each file is read in, compressed, the result is written to memory, read again, uncompressed, and finally the new file size is checked. In every cycle, compress allocates two large byte arrays for input and output; other than that, it allocates very few heap objects. It is believed that compress is not a typical representative of an object oriented application [24]. For our research, the program is not interesting for it does not generate problematic heap shapes.

**db** is the only program in the suite that is not derived from a real-world application. It simulates a simple database management system with a file of persistent records and a list of transactions as inputs. The task is to first build up the database by parsing the records file and then to apply the transactions to this set. Accordingly, db has a very distinct live heap profile: the heap size grows linearly during the building of the database but stays at a fixed level for the entire time of the execution of the transactions. For our research, the program is not interesting for it does not generate problematic heap shapes.

**jack** is a commercial application implementing a parser generator. The program performs 16 iterations of building up a live heap structure and collapsing it again. According to the SPEC documentation, it repeatedly generates a parser from the same input. Because no data survives between iterations, jack can run in a fairly small live heap. For our research, the program is not interesting for it does not generate problematic heap shapes.

**javac** is the JDK 1.0.2 Java compiler iterating three times over several thousand lines of Java code; the source code of jess serves as input for javac. This program’s heap depth grows throughout the compilation phase reaching rather large values so javac is one of the targets of our scalability research.

**jess** is an expert system which reads a list of facts about several word games from an input file and attempts to solve the riddles. For our research, the program is not interesting for it does not generate problematic heap shapes.

**mtrt** is a 3D ray tracer rendering a scene depicting a dinosaur. The program ray traces a picture by dividing the input data into two sections and starting a new working thread for every section. As this program’s heap depth grows rather big, it is one of the targets of our scalability research.
**mpegaudio** is an ISO MPEG Layer-3 audio decoder. This program barely allocates any data and it is not surprising that the heap it generates is very shallow; the program is not interesting for our research.

**SPECjbb2005** is a server-side Java application simulating the three tier client server transaction processing system. While this application is parallel, realistic and has a large working set, it is not interesting for our research because its heap is consistently very shallow.

Dacapo benchmark suite is produced by a DaCapo research project, which was funded by a National Science Foundation. The suite was intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. DaCapo suite is evolving to meet the research community needs and is open source. For this work, we have used the *dacapo-2006-10* release of the suite that comprises a set of eleven non-trivial, real-world, open source Java benchmarks. Only seven of the eleven benchmarks could be successfully run by the version of Jikes RVM we have based our research on and all the results are reported for these seven benchmarks:

- **antlr** parses one or more grammar files and generates a parser and lexical analyzer for each. For our research, the program is not interesting for it does not generate problematic heap shapes.

- **bloat** performs a number of optimizations and analysis on Java bytecode files. This program is one of the targets of our research; its object graph is very deep for some GC cycles.

- **hsqldb** executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application. For our research, the program is not interesting for it does not generate problematic heap shapes.

- **jython** interprets the pybench Python benchmark. For our research, the program is not interesting for it does not generate problematic heap shapes.

- **lusearch** uses lucene to perform a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. For our research, the program is not interesting for it does not generate problematic heap shapes.

- **pmd** analyzes a set of Java classes for a range of source code problems. This application has a very deep object graph at the beginning of the run which becomes rather shallow after a few GC cycles.

- **xalan** transforms XML documents into HTML. This application has the deepest average object graph depth among all the considered benchmarks.
4.2 Prototypes

Our work is implemented in three new flavors of the parallel stop the world mark and sweep collector of Jikes RVM. Most of the modifications were applied to the tracing phase of the basic collector. We have implemented the following new tracing phases:

- Replace the parallel DFS tracing with the single threaded BFS tracing and count the number of elements in each BFS layer of the object graph. This tracing routine was extended to compute the idealized trace utilization measure for every GC cycle during the run.

- Trace through the object graph in a single threaded DFS manner while computing the size and the depth-to-size ratio for every traced subtree. This tracing routine computes the sources and the targets of the additional shortcutting references and incorporates the new references into the objects during the trace.

- Assign some of the tracers to perform the regular trace and some of the tracers to perform the helper trace. Divide the roots among the regular tracers and let them trace as usual with an additional check for the objects traced by the helper tracers. Helper tracers do not get their share of the program roots but pick up their roots and trace form them marking objects in a designated color. This flavor of the tracing phase is the most complex as it requires synchronizing among the working threads.

In addition to the modified tracing phases, we have implemented a framework allowing running the tracing phase several times for each GC cycle. This framework allowed us to perform evaluation of our proposed methods of overcoming the inherent heap shape induced tracing scalability problems. The first time the tracing phase is run, it computes the idealized trace utilization of the heap at the moment the program is stopped for the GC cycle. The second time, the tracing routine performs the method being evaluated. The third, and the last time the tracing phase it run, it performs the idealized trace utilization computation again so that idealized trace utilization before and after the change can be compared. In addition co the idealized trace utilization comparison, the framework allows to collect statistics specific to the method being evaluated, e. g. the number of added references in the shortcuts method.

4.2.1 Modifying the Trace

Tracing collectors perform a transitive closure operation over the set of collected root objects. In MMTk, the collection process is encapsulated in the so called ‘plans’ implemented by the org.mmtk.plan.Plan objects. Tracing behavior is governed by the ‘trace’ objects, separated into the local, org.mmtk.plan.TraceLocal, and global,
Listing 4.1: completeTrace method of org.mmtk.plan.TraceLocal

```java
public void completeTrace() {
    if (!rootLocations.isEmpty()) { //add roots to the working queue
        processRoots();
    }
    do {
        while (!values.isEmpty()) { //while working queue is not empty
            ObjectReference v = values.pop(); //pop object off the working queue
            scanObject(v); //scan the popped object
            //as part of scanning, object’s referents
            //are pushed to the working queue
        }
    } while (!values.isEmpty());
}
```

org.mmtk.plan.Trace, components. Roots are enumerated by a VM-dependent operation and their locations are pushed onto the tracing work queue when VM calls back the routines provided by the org.mmtk.plan.TraceLocal. After the roots are enumerated, collector threads iterate over their work queues, marking objects, enumerating their pointers and enqueueing the discovered objects for scanning (pushing them onto the working queue). The iteration terminates when all the queues are empty. The essence of the tracing phase is captured the a simple completeTrace routine shown in Listing 4.1.

In order to replace the DFS tracing implemented by the completeTrace routine shown in Listing 4.1, we have added a temporary queue and modified the method as shown in Listing 4.2. Root objects are first copied to the temporary queue, then objects are popped of the temporary queue, scanned for references and their referents are put onto the regular working queue. When the temporary queue is empty, all the objects in the first BFS level are scanned and the discovered objects (second BFS level) are stored in the working queue. The process is repeated for every BFS level or till there are no objects in the working queue after emptying the temporary queue. During the process, we collect statistics, like how many objects were discovered at every BFS level and how many objects were discovered and scanned at every moment. Additional profiling information can be collected, like the types of the objects most frequently found in the deep and narrow areas of the object graph.

### 4.2.2 Computing the Idealized Trace Utilization

As described in Section 2.3, in order to compute the idealized trace utilization for the given heap shape, we need to know the amount of objects discovered at every step of the BFS tracing process and the amount of processors performing the trace. We encapsulate the required counters in the DepthClosureCounts class. Instances of
this class are created at the start up, one for every trace phase that composes the GC cycle. Before the trace starts, the object’s counters are initialized. During the trace, counters are updated with every new discovered and scanned object. These numbers are fed into the algorithm computing the idealized trace utilization, partially shown in Listing 4.3. Every time an object is popped off the tracing queue during the completeBFSTrace routine, method addMsPopped of the counting object is called. Knowing how many objects were discovered (pushed) and scanned (popped) at the moment it is called, the method computes amount of work available to be traced assuming 1, . . . , 1024 tracing processors. At the end of the trace, method endClosure is called which computes the utilization, the speedup and the percentage of stalls according to formulae presented in Section 2.3.

4.2.3 Adding Collection Phases

In MMTk, garbage collection cycle proceeds in a structured way through several different phases. If the collector is parallel, there is a need to synchronize collector threads between the phases. MMTk implements the generic code to allow for this phasing and synchronization as well as the default behaviors for the simplest possible collector. Specific collectors can alter the default behavior by overriding the generic functionalities. The implementer can define new collection phases, create new phases ordering, and specify new phase’s behavior. The basic stop the word mark sweep collector of MMTk defines the following phases:

1. initPhase is the complex phase that initializes the cycle. This phase consists in
public void addMsPopped(long pushed, long popped) {
    long work = pushed - popped; //amount of discovered unscanned objects
    long remain;
    int processors;
    for(int i = 0; i < POINTS; i++) { //repeat for every number of processors
        processors = procs[i];
        if(popped == this.lookFor[i]) { //if anticipated amount of discovered objects
            if(work < processors) { //if there is not enough work to keep all the processors busy
                remain = 0; //no remains to be counted
                this.acesses[i] += processors; //count units of work
                this.stalls[i] += processors - work; //count processor stalls
            } else { //if there is enough work to keep all the processors busy
                remain = work%processors; //work remaining to be counted
                this.acesses[i] += work - remain; //count units of work
            }
            this.lookFor[i] = pushed - remain; //when the computation will be repeated for this number of processors
        }
    }
}

public void endClosure(long pushed) {
    int processors;
    for(int i = 0; i < POINTS; i++) { //repeat for every number of processors
        processors = procs[i];
        if(this.acesses[i] != 0) {
            this.util[i] = (pushed * 100)/(this.acesses[i]); //compute utilization
            this.speedup[i] = (pushed * processors)/(this.acesses[i]); //compute speedup
            this.stallop[i] = (this.stalls[i] * 100)/(this.acesses[i]); //compute the percentage of stalls
        }
    }
}
defining the cycle type (some collectors can have different types of collections, like mature and nursery in the generational collectors) and initiating the first collectors handshake.

2. rootClosurePhase is the complex phase that determines objects reachable from roots. This is the main portion of the GC cycle and consists of the following phases:

(a) PREPARE this phase is required to set the stage for the root collection. Both mutators and collectors can participate in this action. In addition, heap spaces can require actions to prepare them for the collection.

(b) prepareStacks this phase prepares mutators’ stacks for the collection. VM specific actions are performed in this phase.

(c) STACK_ROOTS in this phase collector threads compute stack roots.

(d) ROOTS in this phase global and static roots are computed. As an additional optimization, enabled by default in Jikes RVM 3.10, boot image roots are computed as well. This optimization makes a lot more objects available to the tracers as roots so that they do not need to be discovered by a traversal. This makes the root object counts seem unusually large as compared with the numbers obtained by state of the art JVMs. In addition, it causes some objects that otherwise would have to be discovered at the deep levels of the traversal process, to be discovered earlier in the trace. On one hand, this feature can be argued to distort our heap shape related measurements. On the other hand, as all the objects found as boot image root objects are Jikes RVM objects and not application objects, this feature actually assures that the deep and narrow structures we discover belong to the application and not to the VM.

(e) CLOSURE this is the tracing phase of the collector, governed by the completeTrace method shown in Listing 4.1.

3. refTypeClosurePhase this is the complex phase where reference objects (weak, soft and phantom references) are processed. Objects that need to be collected become roots to a new closure phase.

4. completeClosurePhase completes the trace, releases the trace related objects, collects statistics and prepares the heap spaces to resume the run.

5. finishPhase completes the cycle.

In our prototypes, we have disabled reference objects collection to simplify the implementation. The refTypeClosurePhase has became no-op as a result. We have not altered the sequences of phases; the only change was to introduce several modified
closure phases into the single cycle. The basic collector in our framework has the same phases as above but the rootClosurePhase complex phase is replaced with the following sequence of phases:

1. PREPARE unmodified
2. prepareStacks unmodified
3. STACK_ROOTS unmodified
4. ROOTS unmodified
5. prepareClosure new phase required to remember the roots for the sake of the additional tracing phases
6. depthClosure new flavor of the tracing phase which computes the idealized trace utilization of the heap and collects statistics as described in Section 4.2.2
7. prepareClosure prepares for the second trace. This phase restores the roots saved before the previous trace and remembers them for the sake of the next trace. In addition, objects collecting the statistics must be correctly assigned. In the case of parallel trace, this phase must also take care of sharing the roots among the tracing threads.
8. regClosure this is the main closure and is being implemented differently in the collector implementing the shortcuts algorithm and in the collector implementing the colored trace.
9. prepareClosure prepares for the third trace, similar to the earlier preparation phases except that it does not need to save the roots.
10. depthCheckClosure new flavor of the tracing phase which computes the idealized trace utilization of the heap and collects statistics as described in Section 4.2.2. For the shortcuts method, this phase is almost the same as depthClosure. For the colored trace method, the behavior is overridden to compute the idealized trace utilization in a slightly different way explained in Section 5.2.

We replace the regular phases object with our own phases object during the system start-up so the collection cycles are governed by the plan described above allowing us to run the tracing phase 3 times during the single cycle.
4.2.4 Modifying the Object Headers

As described in Jikes RVM documentation, each Java object is composed of the following pieces:

- The JavaHeader. This portion of the object supports language-level functions such as locking, hash codes, dynamic type checking, virtual function invocation, and array length.

- The GCHeader. This portion of the object supports allocator-specific requirements such as mark/barrier bits, reference counts, etc.

- The MiscHeader. This portion supports various other clients that want to add bits or words to all objects. Basically, this is a way to add an instance field to java.lang.Object. Typical uses for the MiscHeader are profiling and instrumentation.

- The object’s instance fields.

The GCHeader has very limited number of bits available for use of the collectors without having to modify a lot of generic code. As our prototypes did not have to be memory or compute efficient, we have decided to use the MiscHeader to keep the per object data required by the prototypes. MiscHeader is created exactly for such use and can be easily extended with new fields of arbitrary length. We have added the declaration for the size of the additional fields in org.jikesrvm.objectmodel.MiscHeaderConstants as well as the field initialization, getter and setter methods in the MiscHeader object. This way, every object allocated by the modified Jikes RVM configuration, had additional pointer sized field in its header as required by our prototypes.

4.2.5 Adding Shortcut References

To implement the shortcuts method, we interpret the new field in the object header as the additional heap reference. Implementation consists of the following changes:

- Initialize the additional object header field to null at object allocation.

- Add new command line options to govern the shortcuts adding algorithm. Jikes RVM has a convenient mechanism for implementing additional command line options and delivering their values to the collector at the initialization time.

- Implement the shortcut adding algorithm. As explained in section 3.1.1, the algorithm computes the size and the depth of the traversed subtree when the subtree node is popped of the tracing queue. To allow for different possible implementations, we have defined the interface, called BalancerInterface, that
defines the required behavior of the method. The interface is presented in Listing A.1 in the Appendix and defines the following methods: initBalance method is called when the trace is started to allow the implementing object to initialize itself; processObject method is called every time the object is popped off the tracing queue, this method computes the properties of the object’s reachability subtree and introduces shortcuts if required; reportBalance() is called when the trace is ended to enable the implementing object to deliver the report about how many shortcuts were added. Specific implementation of the BalancerInterface is presented in Listing A.2 in the Appendix.

• Implement three new flavors for the tracing routine. The first flavor is to modify the scanning routine to null the additional object header field of the scanned objects. This flavor is based on the tracing routine computing the idealized trace utilization measure and is invoked during the first tracing phase, which evaluates heap state before introducing the shortcut references. The change is required to remove the shortcuts introduced by the previous cycle. The second new flavor is to invoke the BalancerInterface method every time the object is popped of the tracing queue. This flavor is used during the second, heap changing tracing phase. The third flavor is to modify the scanning routine to follow the shortcut references by looking into the new field of the MiscHeader of the scanned objects. This flavor is based on the tracing routine computing the idealized trace utilization measure and is invoked during the third tracing phase, which evaluates heap state after introducing the shortcut references.

4.2.6 Tracing Randomly in Parallel

To implement the method of tracing randomly in parallel, more changes to the Jikes RVM were needed. First of all, we interpret the new field in the object header as the special color mark of the tracing phase. Implementation consists of the following changes:

• Initialize the additional object header field to null at object allocation.

• Take care of cleaning the special color marks of the objects between the GC cycles. Mark sweep collectors of MMTk store mark bits either in the GCHandler of the object or in the separate table. The regular collector needs only two colors: objects are allocated in one color and traced using the other color. At the end of the trace, the meanings of the colors are swapped so those new objects are allocated in the same color as the objects survived the previous cycle. Our framework uses more colors because we run three tracing phases: objects are allocated in the first color; the first trace marks the reachable objects with the second color, etc. We use the MMTk option where the regular trace colors are
stored in the GCHeader of the objects. The helper trace colors are stored in the new field we have added to the MiscHeader of the objects. At the end of the cycle, heap objects can have different values in their regular color bits and different values in their additional color word. We add a phase that goes over all the heap objects, collects these values and prepares objects to the continuation of the run. This phase is similar to the traditional sweep. In regular mark sweep collectors of MMTk, the sweep can be done either eagerly or lazily. We choose the eager sweep option and perform the additional colors cleanup during this phase. In addition, this phase is used to collect statistics and perform sanity checks for the algorithm correctness.

- As with the shortcuts method, we add new command line options to govern the new tracing algorithm. These options are explained in Section 3.2: the number of helper threads and the number of the objects that each helper trace marks in a specific color. For technical reason, we have an additional parameter - the number of the colors available for the helpers during the trace. This parameter is required to pre-allocation of the colors table at start up as there is no way to dynamically allocate table entries during the run.

- Implement all the aspects of the algorithm. The color table is implemented in Color.java class presented in Listing A.3 in the Appendix. Static aspects of this class implement the table and individual objects implement individual colors. The class contains locks to synchronize the helper threads accessing the table. Collector threads are assigned to be either the regular or the helper tracers at the startup. Collectors that are designated to be helpers, hold the state encapsulated in the Helper object that is presented in Listing A.4 in the Appendix. As part of the helper trace, the helper picks up the heap object that will be used as a tracing root. Code supporting this function is added into the space implementation. In addition, a simple random number generator was used.

- Implement two new flavors for the tracing routine. The first new flavor is a parallel trace with some threads running the regular and some the helper tracing routine. The second flavor is based on the tracing routine computing the idealized trace utilization measure and is invoked during the third tracing phase. The idealized trace utilization computation is modified in the following way: when the trace encounters an object marked by the additional color, all the objects saved during tracing with this specific color are added to the discovered objects list at once. This is done to evaluate the impact of the additional tracing on the idealized trace utilization measure of the heap trace.
Chapter 5

Results

We have implemented our two prototypes on Jikes RVM version 3.1.0, using the stop-the-world mark-and-sweep collector as the starting point. We have ran the prototypes on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors.

Our research platform is not a many core machine while, as can be seen from the idealized trace utilization data presented in Section 2.4, the scalability problems we aim to resolve start to be significant when there are at least 32 working processors available for the trace. Even if we have had an access to a computer with as many cores, it would be required to have the operating system that capable of supporting this high parallelism along with the Java runtime capable to scale appropriately. Not having such a platform available, we have chosen not to measure the execution time of the prototypes. Instead, we have evaluated the idealized trace utilization measure on the different heap shapes, before and after they were acted upon by our prototypes.

As there were no timing measurements involved, a single run for each measurement point sufficed for the method of adding shortcuts. For the method of tracing through random roots, every measurement point is averaged over 5 runs to account for the lack of determinism resulting from picking roots at random.

5.1 Adding Shortcut References

In this section we report the results of computing the idealized trace utilization measure before and after adding shortcuts as explained in Section 3.1. For each GC cycle, we first measured the idealized trace utilization, then added shortcuts, and finally, measured the idealized trace utilization of the resulting object-graph. For each execution, we have accumulated these values in order to compute the worst and the average values among all the GC cycles. In what follows, we present and analyze the worst case utilization and the average utilization, and report the maximal and the average number of shortcuts that were added to achieve the impact.
Recall that the algorithm described in Section 3.1.1 uses several parameters. We have set the following values: we only added shortcuts to an object whose subgraph has size of at least 50 objects and depth-to-size ratio of at least 0.2. The distance between the shortcut source and the shortcut target (shortcut length) was set to 50 and the distance between two consecutive sources in the same path (shortcut distance) was set to 25. These values were chosen by running the algorithm on a synthetic benchmark with different parameters sets and comparing the results. As will be shown shortly, these values worked well for the majority of our benchmarks. For some benchmarks, however, these values did not produce good results and additional tuning was required.

For benchmarks with no obvious scalability problems, the algorithm did not add shortcuts at all and so there was no change in object-graph properties and in the calculated measure. These benchmarks were: check, compress, jess, db, mpegaudio, jack, hsqldb, and lusearch. For antlr, there were several cycles where the algorithm added a few shortcuts but this had no effect on the already highly scalable heap shape of antlr. The maximal amount of added shortcuts was 16, while the average was less than 10, in a heap of about 230,000 live objects. It was already observed in Section 2.4, that all the above benchmarks show no problematic heap shapes. This result provided a sanity check: our algorithm does not introduce unneeded shortcuts.

For the jython benchmark, almost the same amount of shortcuts were added in all the collection cycles: a maximum of 263 shortcuts and an average of 251. This may seem superficial as jython did not show poor heap shapes in Section 2.4 for up to 512 processors. However, as can be seen in Figures 2.2 and 2.3, for 1024 processors the utilization of jython’s heap shape drops to 82 on average and to 81 in the worst case. Indeed, when shortcuts are added, the utilization improves for this large number of processors. Figure 5.1 shows this improvement graphically. We note here that while jack benchmark shows idealized trace utilization measure values similar to those of jython in Figures 2.2 and 2.3, our algorithm did not add shortcuts in jack. This can be explained by the smaller size of jack benchmark as compared to jython both in terms of running time and the heap size. There are on average about 500 thousands live heap objects in jython while only a 100 thousands live objects in jack; moreover, jython requires 64 MBytes of heap while jack fits comfortably in 16.

Dramatic improvements were obtained for mtrt as shown in Figure 5.2. The maximal number of added shortcuts was 110 and the average was 94, in a heap of about 500 thousands live objects. Excellent improvements were obtained for xalan too, as shown in Figure 5.3. The maximal number of added shortcuts was 888 and the average was 536, in a heap of about 400 thousands live objects. Consistent improvement was achieved for bloat as well, as shown in Figure 5.4. The maximal number of added shortcuts was 940 and the average was 378, in a heap of about 400 thousands live objects.

For the javac and the pmd benchmarks no consistent results were obtained with
Figure 5.1: Worst case object-graph trace utilization before and after adding shortcuts for *jython*.

Figure 5.2: Worst case and average object-graph trace utilization before and after adding shortcuts for *mtrt*.

Figure 5.3: Worst case and average object-graph trace utilization before and after adding shortcuts for *xalan*.
Figure 5.4: Worst case and average object-graph trace utilization before and after adding shortcuts for bloat.

our default set of parameters. It was noted before (see Table 2.3) that these two benchmarks generate deep object-graphs only at some points during the program’s execution. For javac, the live object-graph depth grows throughout the benchmark cycle; for pmd, the live object-graph is very deep at the beginning and remains consistently shallow afterwards. When the heap is large relative to the live objects set, garbage collection tends to be triggered at points where the object-graph is shallow. This is why for these two benchmarks, the average case value of the idealized trace utilization measure is very different from the worst case. With the default set of parameters we could see only a few garbage collection cycles during the run for which improvement was gained. This was not frequent enough to show on the average case, and did not happen for the worst-case cycle. Still, in those rare cycles the depth was reduced by a factor of 10 and the idealized trace utilization measure score was improved.

Since the default set of parameters did not allow improvements, we attempted further tuning. We attempted reducing the amount of added shortcuts (by increasing the shortcut distance). For javac, we increased the minimum subgraph size from 50 to 500, reduced the depth-to-size ratio from 0.2 to 0.1, and increased the shortcut length from 50 to 100. As a result, less shortcuts were introduced: a maximum of 292 shortcuts and an average of 16 shortcuts were introduced in the heap that contained about 383 thousands live objects. In addition, these shortcuts were longer than with the default shortcut length parameter and succeeded in collapsing worst case object graph. In Figure 5.5 we can see that the utilization has improved with the new set of parameters for the worst-case; the average was less affected because the worst case is rare in javac.

A similar tuning was required for pmd. We increased the limit on the subgraph
size to 600, reduced the depth-to-size ratio limit to 0.1, increased the shortcut length to 120, and the shortcut distance to 40. As a result, a maximum of 5,874 shortcuts and an average of 432 shortcuts were introduced in a heap of about 434 thousands live objects, leading to impressive improvement of the worst-case utilization shown in Figure 5.5.

As we see in the example of the javac and pmd benchmarks, it may be possible to achieve better improvements by additional tuning. In general, it would be interesting to investigate the relationships between the algorithm parameters, the amount of added shortcuts and the resulting change in the heap shape. Another interesting question for future research is how to dynamically fit the algorithm’s parameters to the application at hand.

5.2 Tracing Randomly in Parallel

We now turn to reporting the results of our prototype implementation of random tracing by idle threads on the SPECjvm98 and the DaCapo benchmarks. To evaluate the efficacy of this approach, we needed to adapt the idealized trace utilization measure from Section 2 to take into account the extra color tracing. We modified it as follows. Consider the heap at the end of the trace. It contains both reachable and unreachable objects and various connected subgraphs are colored in various colors. Some of these colored components are considered reachable and some unreachable, depending on the reachability status of their color. We now compute the idealized trace utilization measure while treating the additional colors in a special manner. We optimistically assume that all the special tracing by reachable colors was executed.

Figure 5.5: Worst case object-graph trace utilization before and after adding shortcuts for javac and pmd.
before the main trace encountered it. In practice, it is possible that some of it was executed concurrently, but we ignored this possible delay. Under this assumption, when the main trace hits a color, say red, we think of it as if all red objects are added to the trace immediately at no cost, whereas the saved work list of the red color trace is added to the main work list at that same clock tick. Thus, more objects are available to the main trace earlier and the load balancing improves. Given this special color treatment we evaluated the idealized trace utilization measure of a heap in the presence of special color tracing.

To collect the results, for each garbage collection cycle we ran the following three passes. We first evaluated the scalability of the heap shape according to the idealized trace utilization measure as described in Section 2. Next, we ran the main trace on half of the processors and the special color trace on the other half of the processors as explained in Section 3.2. Finally, we ran an evaluation on the obtained heap taking note of the special colors as described above. We therefore obtained the same type of results as for the shortcuts method, showing for each possible number of processors, the improvement in the scalability of the trace.

In Figure 5.6, we report the improvement obtained for mtrt. In fact, the improvement seems negligible for this method. However, when introducing the random choice filter described in Section 3.2.5, the improvements become significant, see Figure 5.7. Thus, for mtrt, picking at random only objects that reference an object of the same type is effective for obtaining improvements with random tracing.

For javac the results were not as good. Without the filter (see Figure 5.8), we could not gain much improvement for the worst-case collection. Moreover, on the average, the scalability deteriorated due to a large amount of dead objects that were traced in vain. With the filter, the situation was a bit better; we did not get deterioration,
Figure 5.7: The change in idealized trace utilization when using random trace for \textit{mtrt}, with the random choices filter described in Section 3.2.5.

but the improvement was negligible as shown in Figure 5.9.

To understand the behavior of the random tracing algorithm, we report some statistics collected during the trace in Tables 5.1 and 5.2. For each benchmark, we report the total number of live objects in the heap and the amount of objects colored by the additional tracers. The latter amount is presented as a percentage of live objects. Since in some cases there are more dead than live objects, the percentage of traced objects can exceed the 100\%, as it sometimes does. To obtain better understanding of the trace, objects colored by the additional tracers are separated into three categories: (1) reachable objects colored by reachable colors, (2) unreachable objects colored by unreachable colors, and (3) unreachable objects colored by reachable colors. The reported information is accurate, since we determine reachability in a separate independent regular trace, while the program is still halted. Work invested into tracing objects of type (1) can be considered \textit{useful} because it saves time for the main trace and parallelizes the trace. Tracing objects of type (2) is \textit{wasted}, since it does not help the main trace and does not harm it. Tracing objects of type (3) is \textit{harmful}, as it creates floating garbage and can impose additional work on the main trace because the main trace continues to trace from where the helper threads finished. As can be seen in the tables, large percentages of floating garbage created by additional tracers is the main problem we encountered with \textit{javac}.

The tests were run with a trace-limit of 1000, i.e., the number of objects colored by any color does not exceed a thousand. The results are produced for every GC cycle. For the statistics, we then computed the average values for the run. To account for the algorithm’s non determinism resulting from random object picking, we ran each benchmark five times and averaged the results. In Table 5.1 we report the results obtained without the filter and in Table 5.2 – with the filter. It can
Figure 5.8: The change in idealized trace utilization when using random trace for javac when no filter was used. While there were a lot of GC cycles with improved utilization, including the worst case, the average utilization became worse due to many dead recursive structures traced by the special trace in other GC cycles.

Figure 5.9: The change in idealized trace utilization when using random trace for javac, with the random choices filter described in Section 3.2.5.
<table>
<thead>
<tr>
<th>name</th>
<th>live objects (thousands)</th>
<th>useful helpers work (%)</th>
<th>wasted helpers work (%)</th>
<th>floating garbage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>384</td>
<td>3.99</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>jack</td>
<td>110</td>
<td>11.7</td>
<td>0.16</td>
<td>0.072</td>
</tr>
<tr>
<td>javac</td>
<td>316</td>
<td>9.3</td>
<td>0.16</td>
<td>357.6</td>
</tr>
<tr>
<td>jess</td>
<td>128</td>
<td>11.9</td>
<td>0.17</td>
<td>0.12</td>
</tr>
<tr>
<td>mtrt</td>
<td>363</td>
<td>8.69</td>
<td>0.72</td>
<td>12.52</td>
</tr>
<tr>
<td>antlr</td>
<td>220</td>
<td>9.9</td>
<td>0.95</td>
<td>7.14</td>
</tr>
<tr>
<td>bloat</td>
<td>380</td>
<td>7.94</td>
<td>0.05</td>
<td>0.07</td>
</tr>
<tr>
<td>hsqldb</td>
<td>3,360</td>
<td>4.11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jython</td>
<td>128</td>
<td>3.31</td>
<td>0.04</td>
<td>0.023</td>
</tr>
<tr>
<td>lusearch</td>
<td>230</td>
<td>9.16</td>
<td>0.08</td>
<td>0.17</td>
</tr>
<tr>
<td>pmd</td>
<td>370</td>
<td>7.25</td>
<td>0.41</td>
<td>37.24</td>
</tr>
<tr>
<td>xalan</td>
<td>330</td>
<td>5.89</td>
<td>0.07</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Table 5.1: Properties of the random trace with a trace-limit of 1000 when no filter is used: overall number of live objects, the percentage of live objects discovered by random trace, the percentage of dead objects traced by the helper threads, and the percentage of objects that became floating garbage due to the random trace.
Table 5.2: Properties of the random trace. This table is similar to Table 5.1 except that the results were obtained while using the random choices filter described in Section 3.2.5.

<table>
<thead>
<tr>
<th>name</th>
<th>live objects (thousands)</th>
<th>useful helpers work (%)</th>
<th>wasted helpers work (%)</th>
<th>floating garbage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>384</td>
<td>10.8</td>
<td>0.036</td>
<td>0.004</td>
</tr>
<tr>
<td>jack</td>
<td>110</td>
<td>25.8</td>
<td>0.08</td>
<td>0.024</td>
</tr>
<tr>
<td><strong>javac</strong></td>
<td>316</td>
<td>14.61</td>
<td>0.14</td>
<td>27.01</td>
</tr>
<tr>
<td>jess</td>
<td>128</td>
<td>30.22</td>
<td>0.21</td>
<td>0.045</td>
</tr>
<tr>
<td><strong>mtrt</strong></td>
<td>363</td>
<td>18.25</td>
<td>2.31</td>
<td>0.92</td>
</tr>
<tr>
<td>antlr</td>
<td>220</td>
<td>13.93</td>
<td>1.35</td>
<td>0.13</td>
</tr>
<tr>
<td><strong>bloat</strong></td>
<td>380</td>
<td>17.58</td>
<td>0.13</td>
<td>0.2</td>
</tr>
<tr>
<td>hsql</td>
<td>3,360</td>
<td>4.8</td>
<td>0.023</td>
<td>0</td>
</tr>
<tr>
<td>jython</td>
<td>128</td>
<td>12.2</td>
<td>0.1</td>
<td>0.03</td>
</tr>
<tr>
<td>lusearch</td>
<td>230</td>
<td>20.75</td>
<td>0.78</td>
<td>0.467</td>
</tr>
<tr>
<td><strong>pmd</strong></td>
<td>370</td>
<td>14.39</td>
<td>0.58</td>
<td>51.24</td>
</tr>
<tr>
<td><strong>xalan</strong></td>
<td>330</td>
<td>15.74</td>
<td>0.11</td>
<td>0.98</td>
</tr>
</tbody>
</table>

be seen that the filter, i.e., the bias towards objects that can reference their own type, typically increases the percentage of useful work by the tracers and reduces the amount of floating garbage. The latter is especially noticeable for javac. But it is not deterministic. For pmd the floating garbage actually increased with the use of the filter.
Chapter 6

Conclusion

As garbage collected languages remain highly desirable and as the amount of hardware parallelism is steadily rising, the question of garbage collection heap tracing scalability becomes acute. This work provides a first investigation of the heap-tracing scalability. The research started by investigating object graphs generated by a set of standard Java benchmarks. When it became clear that there exist benchmarks with very deep and narrow live objects graphs, we proceeded to devise a way to quantify the amount of tracing scalability allowed by the object graph. As a result, we have proposed an idealized trace utilization measure which captures the object graph scalability properties while being independent of the load balancing between the garbage collection threads, memory access latencies and variability of the object scanning times. We have demonstrated that poor idealized trace utilization scores are obtained for Java benchmarks with the deep and narrow live object graphs. Such heap shapes can foil the scalability of the application on highly parallel platforms, due to the non-scalability of the garbage collection tracing activity during the execution.

We then investigated two possible directions for ameliorating the problem: adding heap shortcuts and tracing on idle processors. In our measurements, heap shortcuts method is more effective. However, further investigation is required to validate the use of shortcuts within a practical garbage-collected system. For such a research, a larger manycore system and a scalable parallel OS and JVM runtime are required. Even though we did not have access to such a platform, we believe that preparing the ground for high parallelism is an important goal of system research. In particular, we believe that preparing memory management today for a potential highly-parallel platforms that may arrive tomorrow is an important research goal.
Appendix A

Computer Programs

A.1 Adding Shortcut References Code

Listing A.1: BalancerInterface.java

```java
package org.mmtk.plan.scale.balance;

//removed -- imports

/**
* Used to abstract the process of balancing object graph
* by introducing new references into individual objects.
*
* The process can be implemented by different
* concrete algorithms
*
* @author kathy
*
* */
@Uninterruptible
public interface BalancerInterface {

/**
* Called before the balancing process over the heap
* Balancer can initialize or reset counters and structures
*
*/
public void initBalance();

/**
* Analyze individual object encountered during trace
* and include it in the heap balancing process.
* Called by the tracer just after the object was popped off
* the stack and scanned for references.
*
* @param objectRef object reference to process
* @param level level in trace stack or distance from root
* @param children number of children encountered in this object scan
*
*/
public void processObject(ObjectReference objectRef, long level, int children);

/**
* Report the results of the balancing process.
* Writes to the log
*
*/
public void reportBalance();
}
```
package org.mmtk.plan.scale.balance;

//removed -- imports
@Uninterruptible
public class TraceBranchBalancer implements BalancerInterface {
    //constants
    private static final int HISTORY_SIZE = 50000;

    //structures
    private ObjectReferenceArray objectRefs;
    private ObjectReferenceArray candidateRefs;
    private long[] candidateDepth;
    private long[] level;
    private int[] children;
    private long[] descendants;
    private long[] toBottom;

    //counters
    private int size;
    private long lost;
    private long setPointers;
    private long maxDepth;

    TraceBranchBalancer() {
        //removed -- create the fields
    }

    @Inline
    public void initBalance() {
        //removed -- just initialize the fields
    }

    @Inline
    public void processObject(ObjectReference objRef, long objLevel, int objChildren) {
        int parentIndex = findParent(this.size, objLevel);
        if(parentIndex == -1) {
            //root object’s descendants tree is exhausted, process it now and remove all slots
            processSubtree(0, this.size - 1);
            if(objChildren > 0) {
                add(objRef, objLevel, objChildren);
            }
            return;
        }
        if(objChildren == 0) {
            //we do not keep such nodes in tracker, just update their parent
            addLeafToNode(parentIndex);
            if(!isLast(parentIndex)) {
                processSubtree(parentIndex+1, this.size - 1);
            }
            return;
        }
        if(objChildren > Options.sbMaxRefs.getValue()) {
            //we do not keep such nodes in tracker, just update their parent
            addToNode(parentIndex, objChildren);
            return;
        }
        if(!isLast(parentIndex)) {
            processSubtree(parentIndex+1, this.size - 1);
        }
        add(objRef, objLevel, objChildren);
    }

    /**
     * Finds the index where the parent (or the closest ancestor) of this node is stored
     * If the parent is not in the table, this node is not an descendant
     * of the currently tracked nodes. In such a case, all the table has to be processed and
     * emptied.
     * @param objLevel - current node’s level in the global stack
     * @return parent/ancestor index in table, -1 if not found
     */
    @Inline
    private int findParent(int index, long objLevel) {
        int result = -1;
        index = index - 1;
        while(index >= 0) {
            if(level[index] <= objLevel) {
                result = index;
                index = index - 1;
            } else {
                index = index - 1;
            }
        }
        return result;
    }

    public void processObject(ObjectReference objRef, long objLevel, int objChildren) {
        int parentIndex = findParent(this.size, objLevel);
        if(parentIndex == -1) {
            //root object’s descendants tree is exhausted, process it now and remove all slots
            processSubtree(0, this.size - 1);
            if(objChildren > 0) {
                add(objRef, objLevel, objChildren);
            }
            return;
        }
        if(objChildren == 0) {
            //we do not keep such nodes in tracker, just update their parent
            addLeafToNode(parentIndex);
            if(!isLast(parentIndex)) {
                processSubtree(parentIndex+1, this.size - 1);
            }
            return;
        }
        if(objChildren > Options.sbMaxRefs.getValue()) {
            //we do not keep such nodes in tracker, just update their parent
            addToNode(parentIndex, objChildren);
            return;
        }
        if(!isLast(parentIndex)) {
            processSubtree(parentIndex+1, this.size - 1);
        }
        add(objRef, objLevel, objChildren);
    }

    private int findParent(int index, long objLevel) {
        int result = -1;
        index = index - 1;
        while(index >= 0) {
            if(level[index] <= objLevel) {
                result = index;
                index = index - 1;
            } else {
                index = index - 1;
            }
        }
        return result;
    }
}
result = index;
break;
} }
index--;}

return result;

@Inline
private void processSubtree(int from, int to) {
if(from > to) {
    return;
}
//go up accumulating counters in subtree
int index = to;
while(index >= from) {
    int parent = findParent(index, this.level[index]);
    if(parent == -1) {
        break;
    }
    addSubtreeToNode(parent, index);
    index--;
} //remember subtree objects
int distanceParam = Options.sbDistance.getValue();
long toRemember = (this.toBottom[from] > distanceParam) ? distanceParam : this.toBottom[from];
for(int i = 0; i < toRemember; i++) {
    rememberObject(from+i);
}
//analyze subtree dimensions, add pointers if needed
long subtreeSize = this.descendants[from];
double subtreeRatio = (double)this.toBottom[from]/this.descendants[from];
if(subtreeSize >= Options.sbSize.getValue() && subtreeRatio > Options.sbRatio.getValue()) {
    addCandidatePointers(from, to);
} //discard subtree objects
discardSubtree(from);

@Inline
private void rememberObject(int index) {
long totalDepth = this.toBottom[index] + index + 1;
int target = index - Options.sbDistance.getValue();
if(target > 0) {
    if(this.candidateRefs.get(target) == ObjectReference.nullReference() ||
        this.candidateDepth[target] < totalDepth) {
        //replacing candidate ref
        this.candidateRefs.set(target, this.objectRefs.get(index));
        this.candidateDepth[target] = totalDepth;
    } else {
        //forget this object
    }
}

@Inline
private void addCandidatePointers(int from, int to) {
int distanceParam = Options.sbDistance.getValue();
int skipParam = Options.sbSkip.getValue();
int index = from;
//look for the set candidate
while(index < to) {
    while(this.candidateRefs.get(index) == ObjectReference.nullReference() && index < to) {
        index++;
    }
    if(index < to) {
        ObjectReference obj = this.objectRefs.get(index);
        ObjectReference add = this.candidateRefs.get(index);
        VM.objectModel.setShortcut(obj, add);
        this.setPointers++;
    }
    index += skipParam;
} //add new
int target;
while((target = index - distanceParam) >= from) {
    if(this.candidateRefs.get(target) == ObjectReference.nullReference()) {
        ObjectReference obj = this.objectRefs.get(target);
        ObjectReference add = this.objectRefs.get(index);
        VM.objectModel.setShortcut(obj, add);
        this.setPointers++;
    }
}
```java
private void discardSubtree(int from) {
if (from == 0) {
    if (this.toBottom[from] > this.maxDepth) {
        this.maxDepth = this.toBottom[from];
    }
    int index = this.size - 1;
    while (index >= from) {
        discardObject(index);
        index--;
    }
    this.toBottom[from] = 0;
}
}

private void discardObject(int index) {
    //removed -- just nullify the entry
}

private void addLeafToNode(int node) {
    this.descendants[node]++;
    if (this.toBottom[node] == 0) {
        this.toBottom[node] = 1;
    }
}

private void addToNode(int node, int children) {
    this.descendants[node] += children;
    if (this.toBottom[node] == 0) {
        this.toBottom[node] = 1;
    }
}

private void addSubtreeToNode(int node, int sub) {
    this.descendants[node] += this.descendants[sub] + 1;
    long toBottom = this.toBottom[sub] + 1;
    if (this.toBottom[node] < toBottom) {
        this.toBottom[node] = toBottom;
    }
}

private boolean isFull() {
    return this.size == HISTORY_SIZE;
}

private boolean isLast(int index) {
    return this.size == index + 1;
}

// push assuming not full
private void add(ObjectReference objectRef, long level, int children) {
    if (isFull()) {
        this.lost++;
        return;
    }
    objectRefs.set(this.size, objectRef);
    this.level[this.size] = level;
    this.children[this.size] = children;
    this.descendants[this.size] = 0;
    this.toBottom[this.size] = 0;
    this.size++;
    if (isFull()) {
        overflow();
    }
}

private void overflow() {
    this.discardSubtree(0);
}
```
A.2 Tracing Randomly in Parallel Code

package org.mmtk.plan.scale.colors;
//removed -- imports
/**
 * This class is responsible to hold the global color table and synchronize access to it.
 * Class instances represent specific colors with their properties and methods.
 * Index into the colors table is the value of the color mark - that’s why we do not use index=0;
 * Object with color mark zero is unmarked object.
 */
@Uninterruptible
class Color {
    private final static int SAVE_OTHER_SPACE = 50;
    /**
     * Static fields and methods
     */
    private static final Lock colorsTableLock;
    //color table
    public static ObjectReference tableRef;
    private static Color[] colorsTable;
    private static int colorsTableSize;
    private static int colorsTableIndex;
    //trace completion
    private static int traceCompletionIndex;
    private static boolean inTraceComplete;
    private static int colorTableTraces;
    //parameters
    private static int maxObjectsToColor;
    static {
        colorsTableLock = VM.newLock("colorsTableLock");
    }
    /**
     * Called upon plan initialization when parameters are set
     */
    @Interruptible
    static void init() {
        //removed -- create and initialize the fields
    }
    //removed -- lock and unlock routines, assorted simple getters and setters, both static and instance
    private static int getColorsTableEnd() {
        //removed -- compute the last index
    }
    /**
     * Called upon GC cycle start - to clean up previous cycle data
     */
    static void prepareColorsTable() {
        //removed -- initialize the fields before the cycle
    }
    /**
     * Called by helpers when they need to start coloring on the new color
     * @return Color object to use for the speculative trace
     */
    static Color obtainColor() {
        Color result = null;
        if(colorsTableIndex == colorsTableSize) {
            return result;
        }
        acquireColorsTableLock();
        if(colorsTableIndex < colorsTableSize) {
            colorsTableIndex++;
            if(colorsTableIndex < colorsTableSize) {
                result = colorsTable[colorsTableIndex];
            }
        }
        releaseColorsTableLock();
        return result;
/**
 * Called while tracing through abandoned fronts
 * @return Color object to use for the completion trace
 */
static Color obtainColorToComplete() {
    Color result = null;
    Color color;
    int end = getColorsTableEnd();
    acquireColorsTableLock();
    if (!inTraceComplete) {
        inTraceComplete = true;
    }
    if (traceCompletionIndex >= end) {
        traceCompletionIndex = 0;
    }
    while (traceCompletionIndex < end) {
        traceCompletionIndex++;
        if (traceCompletionIndex < end) {
            color = colorsTable[traceCompletionIndex];
            if (color.live && color.numSavedValues > 0 && !color.savedValuesTraced) {
                result = color;
                color.savedValuesTraced = true;
                break;
            }
        }
    }
    releaseColorsTableLock();
    return result;
}

/**
 * Called when getting abandoned fronts before checking trace
 * @param traceLocal
 * @return the number of values
 */
public static long getAllSavedValues(ScaleTraceLocal traceLocal) {
    Color color;
    int numSavedValues;
    ObjectReferenceArray savedValues;
    ObjectReference objRef;
    int i;
    int j;
    long total = 0;
    int end = getColorsTableEnd();
    for (i = 1; i < end; i++) {
        color = colorsTable[i];
        numSavedValues = color.numSavedValues;
        if (color.live && numSavedValues > 0) {
            savedValues = color.savedValues;
            for (j = 0; j < numSavedValues; j++) {
                objRef = savedValues.get(j);
                traceLocal.processNode(objRef);
                total++;
            }
            color.numSavedValues = 0;
        }
    }
    return total;
}

static boolean traceLiveColor(int colorIndex) {
    boolean result = false;
    Color color = colorsTable[colorIndex];
    int end = getColorsTableEnd();
    for (int j = 1; j < end; j++) {
        if (j == colorIndex) {
            continue;
        }
        if (!color.otherColors[j]) {
            continue;
        }
        if (colorsTable[j].markLive()) {
            if (!result) {
                result = true;
            }
        }
    }
    return result;
}
return result;
}

//single threaded
static boolean checkColorsTable(boolean all) {

boolean result = false;
int end = getColorsTableEnd();
for (int i = 1; i < end; i++) {
    Color color = colorsTable[i];
    if (!color.isLive() && color.liveColored > 0) {
        if (all || color.numSavedValues > 0 || color.otherColors() > 0) {
            // this is to fix possible bug related to tracing other space objects
            color.markLive();
            result = true;
            break;
        }
    }
}
return result;
}

//single threaded
static boolean traceLiveColors(boolean withCheck) {

colorTableTraces++;

boolean result = false;
if (colorsTableSize == 0) {
    return result;
}

boolean done = false;
int end = getColorsTableEnd() - 1;

while (!done) {
    done = true;
    for (int i = end; i > 0; i--) {
        Color color = colorsTable[i];
        if (traceLiveColor(i)) {
            result = true;
            done = false;
        }
    }
}

if (!withCheck) {
    return result;
}

if (!result) {
    if (checkColorsTable(false)) {
        return true;
    } else {
        return checkColorsTable(true);
    }
}
return result;
}

/**
 * Instance fields and methods
 */
private final int index;
private Address root;
private long madeLive;
private long colored;
private long liveColored;
private long markedColored;
private boolean live;
private boolean done;
private boolean savedValuesTraced;
private boolean[] otherColors;
private ObjectReferenceArray savedValues;
private int numSavedValues;
private int numSavedDead;
private int numOSObjects;
private int numSavedOSObjects;
private ObjectReferenceArray osObjects;

Color(int index) {
    // removed -- create and initialize the fields before the cycle
}

private void prepare() {

void saveObjectsFront(ObjectReferenceDeque values) {
    int index = 0;
    int lost = 0;
    int size = savedValues.length();
    int objectMark;
    while(!values.isEmpty()) {
        ObjectReference object = values.pop();
        if(index < size) {
            savedValues.set(index, object);
            index++;
            if (Space.isInSpace(SColors.SCALE_MS_SPACE, object)) {
                objectMark = SColors.scaleSpace.getColor(object);
                if(objectMark == 0) {
                    this.numSavedDead++;
                }
            }
        } else {
            lost++;
        }
    }
    this.numSavedValues = index;
    this.done = true;
}

boolean markLive() {
    if (live) {
        return false;
    }
    acquireColorsTableLock();
    live = true;
    releaseColorsTableLock();
    traceLiveColor(this.index);
    return true;
}

void addDependentColor(int otherColor) {
    if(otherColor < 0 || otherColor > colorsTableIndex) {
        if(!this.otherColors[otherColor]) {
            this.otherColors[otherColor] = true;
        }
    }
}

void addColored(int regColor) {
    this.colored++;
    if(regColor == 1) {
        this.liveColored++;
    } else if(regColor == 2) {
        this.markedColored++;
    } else if(regColor == 0) {
        this.deadColored++;
    }
    if(this.colored == maxObjectsToColor) {
        done = true;
    }
}

Listing A.4: Helper.java
//removed - create and initialize the fields

//removed -- lock and unlock routines
static void prepareHelpersTable()
{
  //removed - initialize the fields before the cycle

static Helper getHelper(boolean primary) {
    Helper result = null;
    if (!primary) {
      acquireHelpersLock;
      if (helpersTableIndex < helpersTableSize) {
        helpersTableIndex++;
        if (helpersTableIndex < helpersTableSize) {
          result = helpersTable[helpersTableIndex];
        }
      }
      releaseHelpersLock;
    }    
    return result;
}

/**
 * Private fields and methods
 */
private ScaleTrace helperPrivateTrace;
private CMHelperTraceLocal helperPrivateLocalTrace;
Helper(){
  this.helperPrivateTrace = new ScaleTrace(SColors.metaDataSpace);
  this.helperPrivateLocalTrace = new CMHelperTraceLocal("Helper.localTrace", this.helperPrivateTrace);
}
private void prepare() {
  this.helperPrivateTrace.prepare();
  this.helperPrivateLocalTrace.prepare();
}

void help() {
  if (!done) {
    Color color = Color.obtainColor();
    if (color != null) {
      this.helperPrivateLocalTrace.helperTrace(color);
    } else {
      acquireHelpersLock;
      if (!done) {
        done = true;
      }
      releaseHelpersLock;
    }
  }
}
References


ミיקבול איסוף מידע אוטומטי למשרר רשת
של معدلים

גטי ברבך
מיקבולים איסוף תכונת אוטומטי למספר רב
של מתבדים

הことがある על מחק

לשם مليי חלקי של הדרישה על תכלית ההאר
منهجי למדעי
מודעי המתחב

קטיב ברבש

ווש לסתנ הפטנור — מוכן טכנולוגי לישראל
חג מון תשע”א
אוקטובר 2010
письומ על מחקר שעשה בדר McCart פינט, חבר אורות פטריך
פקרטיא לאمدار הפרש ב
تكצירי

בשにくס האתגרות אוג עדכני לשניים מחמאותיים בולטים מה ancor. מחז. הכל שלמה
מורחבת מעברית יכלה להפתיע, והחברה ייבנ תהליכים שונים שבהם אחדilan את חיבור
נצלון התוכן בחרותה שבтверждаuder היחה. באז' accur, עליים י숏ים המחושב рождения הפרק
ויהיר מרבד. טייטגר, אמצעים המרחורות בכם יצור כולם מהמכס של יד יוזר
ויהיר פופ המ maxHeight לнем שימשים מחמאות יסודים בשירות מרחפת湾יד.
אירופה הק든지 במלאי הזיפיחה של קמצינו שכרתוב מחומצתיות של כלァימשל בשטח
הקורות הכזייה פקט מגרות ג yapt יצור הפרים והפרסים שdehyזים שdehyזים
ליכרה על גמרו מדועים מחומצה ושעמל מזמנעל.

shallאלה השאלת האות יฝรั่งוק אפסי האוסף, הביאו גלק ח纠ב שכרתוב ו▸יוחר, מסל.
גלחело מדועי עונمعنى מחומצה עוזר. גיז שיווקים ינוגים כל המחוב מחומצה
שהייוותים יכל חתפוע על המלך שיילסיק הליול מחומצה מחומצתיות בחוליה. במרג.
מנון אריכים תורם יגבי האובייקטים العدوים לולוס ת🙌חרת הזיפי והייד אוושם תⓣור
ל süre_donת אך הגנה מעובד בולל ברמנך קך להיזד איא חומצה נטמנת.
_SaveD.lock את הקורס יcherche המלועש
בגבודה הגרה את הקורס את השאלת האות יעור פעל ת不得转载
הsaving התלופרות המקובלת לוחך בבל. כי- Java- hå계יפיל שלוח הור.
יחסיומם השמונשפשות נקראה דילב מבהיק שלל דנוזות והתקני
הלקצין והמקובלים מצויקת ויתונשה.

Arishel שפוג מועך על דרג האובייקטים התרבויות של ידית התוכן ה-
בוחן: Java, C-sharp, "Jikes R VM" benchmarks-ו.

Arishel שפוג מועך על דרג האובייקטים התרבויות של ידית התוכן ה-
בוחן: Java, C-sharp, "Jikes R VM" benchmarks-ו.
 Idealized Trace Utilization