Tracing Garbage Collection on Highly Parallel Platforms [*]

Katherine Barabash
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
kathy@cs.technion.ac.il

Erez Petrank [†]
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
erez@cs.technion.ac.il

Abstract

The pervasiveness of multiprocessor and multicore hardware and the rising level of available parallelism are radically changing the computing landscape. Can software deal with tomorrow's potential higher parallelism? In this paper we study this issue from the garbage collection perspective. In particular, we investigate the scalability of parallel heap tracing, which stands at the core of the garbage collection activity. Heap shapes can be sequential in nature, and prevent the collector from scaling the trace. We start by proposing the idealized trace utilization as a scalability measure for evaluating the scalability of a given heap shape. We then examine standard Java benchmarks and evaluate the existence of non-scalable object-graph shapes in their execution. Next, we propose and implement a prototype of garbage collection techniques that attempt to ameliorate the object-graph shape problem. Finally, we measure and report their efficacy.

Categories and Subject Descriptors D.1.5 [*Object-oriented Programming*]: Memory Management; D.3.3 [*Language Constructs and Features*]: Dynamic storage management; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage Collection

General Terms Languages, Performance, Algorithms.

Keywords Runtime systems, Memory management, Garbage collection, Parallel garbage collection.

1. Introduction

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 [33], 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.

Managed programming languages such as Java and C# are becoming the main vehicle for writing large software projects. They provide built-in security, threading support, impressive multi purpose standard class libraries, and last but not least, garbage collection (GC). In order to maintain the advantages of managed languages, it is crucial that applications written in these languages be efficient and scale well on modern and future platforms. In particular, runtime systems including the garbage collection must be made adequately scalable to support efficient execution on future platforms. A vast literature on designing parallel and concurrent garbage collection exists (e.g., [31, 32, 14, 9, 16, 15, 19, 17, 18, 28, 11, 23, 25, 4, 21, 3, 6]); however, the available techniques can probably not fit a highly parallel platform 'as is'. Moreover, even if the garbage collection algorithms scaled perfectly, a new problem arises, because the shape of the object-graph becomes more dominant in foiling scalability. Imagine a program that employs a large linked list. A traversal of such a list is sequential in nature and foils tracing scalability. Other (less extreme) object-graph shapes may be detrimental to parallelism as well.

Understanding the scalability of heap tracing is interesting in general, but it is particularly important in the context of real-time garbage collection [5, 13, 26]. In this area, it is crucial to compute bounds on the progress of the collector and make sure it terminates on time, before the heap gets exhausted. Otherwise, the program may get stuck while trying to allocate on a full heap that has not been garbage collected on time.

A question that naturally arises is whether this problem of heap shapes that are detrimental to 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 facilitate scalability of the garbage collector for such programs? If we cannot solve this problem, then the scalability of the Java and the C# runtimes becomes questionable.

This paper initiates a rigorous investigation of the heap tracing scalability issue. First, since many-core machines are not yet available on the market, we start by proposing an idealized trace utilization measure that, given an object-graph shape, evaluates the amount of parallelism it enables. The 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 benchmarks. We

[*] Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).

[†] Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).

then propose, implement a prototype, and evaluate a couple of 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 let the collector (or the compiler) add pointers to the the headers of objects, to artificially modify the object-graph shape and make it more scalable. Details appear in Section 4.1. The second solution is to let 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. Details appear in Section 4.2.

We have implemented a prototype, to evaluate the potential effectiveness of each solution, on Jikes RVM [2] using the MMTK [8]. We ran measurements on the SPECjvm98 benchmark suite, the SPECjbb2005 benchmark, and the DaCapo benchmark suite [1, 7]. We present and discuss these results in Section 5.

We could not check the actual benefits on a many-core platform, because many-cores cannot be (practically) obtained today. Nevertheless, 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 highly-parallel platform that may arrive tomorrow is an important research goal.

Organization In Section 2 we define and study heap depths of standard Java benchmarks. In Section 3 we describe the idealized trace utilization measure we have devised to quantify object-graph scalability. In Section 3.1 we present scalability measurements for a variety of widespread Java benchmarks, and demonstrate that some of these benchmarks generate object-graphs with poor scalability. In Section 4 we describe two possible solutions. Implementations and results are discussed in Section 5. Related work is reviewed in Section 6, and we conclude in Section 7.

2. Preliminaries: Heap Depth and Tracing

Garbage collectors (GCs) trace objects in the heap to discover which ones are reachable. Tracing collectors, either mark-sweep or copying collectors, trace the heap in order to identify all the live objects; whereas reference counting collectors trace the transitive closure of dead objects whose reference counts drop to zero. This work concentrates on tracing collectors, often employed in today's large systems. These collectors trace all objects reachable from a well-defined set of pointers called *roots*. For details on classical garbage collectors, see [24].

The process of tracing live objects is iterative (or recursive). It starts from a list of objects directly reachable from the roots. Then, each object in the list is marked and each of the unmarked objects directly reachable from it is added to the list. The list of pointers is typically managed as a queue or a stack, creating a BFS or DFS traversals of the live objects respectively. A DFS traversal is usually considered more cache friendly [29], whereas a BFS traversal is more scalable, because more paths are discovered early on and better distribution of work is enabled.

By definition, each reachable object in the heap has one or more paths of pointers starting from a root and leading to it. The length of the shortest such path is defined as the *depth* of the object or its *distance from the roots*. The depth of the entire live object-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 in which each object is traced in one single computation step, it still holds that using P processors to trace a heap in which the live object-graph has L live objects and depth D requires time of at least $max\{L/P, D\}$ computation steps.

As a first step in our investigation we measured the object-graph depth for all benchmarks in the following benchmarks suites: SPECjvm98 [1], SPECjbb2005 [1], and DaCapo [7]. These results extend previous work [20, 30], which provided similar measurements only for SPECjvm98 and pBob (a work-constrained version of SPECjbb2000). In order to perform the measurements, we changed the mark and sweep stop-the-world garbage collector of Jikes RVM to measure the live object-graph depth, and triggered a garbage collection very frequently during the run – after every 32 KBytes of allocation.

This step gave us a first glimpse into the existence of deep live object-graphs in typical benchmarks, and furthermore, on how object-graph shapes change during the execution of the benchmarks. It turned out that several benchmarks (*javac*, *raytrace* and *mtrt* of *SPECjvm98* and *bloat*, *pmd* and *xalan* of *blac06*) exhibit deep and narrow forms of live object-graphs. We note that these results are consistent with the partial information provided by previous work [30, 34]. An additional observation we could make was that non-scalable live shapes appeared consistently during the runs of some benchmarks (*mtrt* and *xalan*), but only occasionally in the run of others. These latter applications exhibited life cycle dependent patterns. For example, in *javac* and *bloat*, the depth of the object-graph increases consistently through 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 scalable object-graph and we do not study it further in this paper.

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-object shape is not scalable.

We proceeded by executing the 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 different object-graph depths. In Table 1 we present the maximal and the average object-graph depths for *SPECjvm* and *blac06* 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. To reduce the amount of presented data, we

name	heap size (MBytes)	GC cycles	max depth	avg depth
db	32	3	16	14
jack	16	34	38	36
javac	32	15	**1,234**	**231**
jess	16	63	32	29
mtrt	32	8	**1,416**	**1,413**
antlr	32	16	60	30
bloat	48	344	**1,195**	**352**
hsqldb	128	6	47	38
jython	64	49	128	124
lusearch	64	65	38	14
pmd	48	59	**18,482**	**362**
xalan	128	129	**8,476**	**4,199**

Table 1. Number of GC cycles and maximal and average object-graph depths for the SPECjvm98 and the DaCapo benchmark suites; and the heap size used to obtain the results with the normally triggered GC cycles.

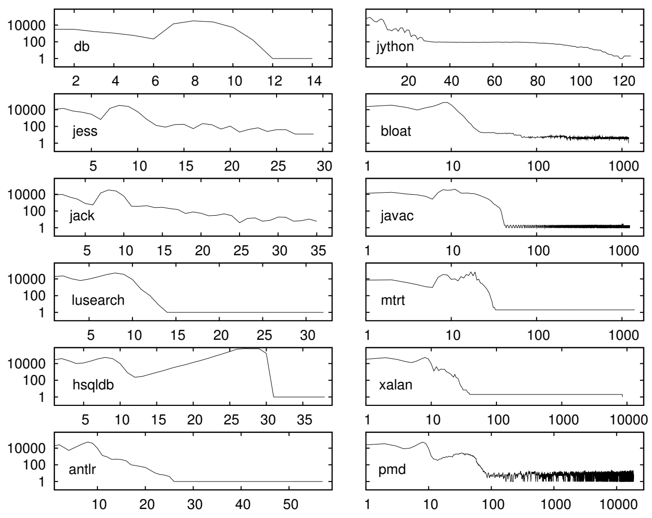

Figure 1. Object distribution among the different depths. The x-axis represents the depth and the y-axis represents the number of objects found at that depth.

have omitted data for *check*, *compress* and *mpegaudio* from all the tables and figures that follow. These benchmarks are very small in terms of their heap usage and their object-graphs are typically shallow. Data for benchmarks that show non-scalable live object-graph shapes and that will be discussed further in this paper is emphasized in bold typesetting.

In Figure 1, we show the distribution of objects depths when the object graph is the deepest encountered by a normally triggered GC cycle. 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 objects' depths for *javac*, *mtrt*, *bloat*, *pmd* and *xalan* benchmarks. For example, in the heap built by *xalan*, the majority of the objects are of depth at 17 or less. Looking at higher depths, the shape of the object-graph becomes more and more narrow, until, starting from depth 39 there are only two objects at each depth.

The object-graph depth parameter 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 to identify their scalability.

3. Idealized Trace Utilization Measure

Next, we propose a measure that expresses the scalability of a given live object-graph with respect to a given number of processors. In the remainder of this paper, we will denote by *heap shape* the shape of the live object-graph in the heap. To better understand the scalability of heap shapes, we consider a simplified and clean version of a trace when run with a given number of processors. Recall that in a trace we maintain a list of objects to be scanned and iteratively pick one, mark it, and insert its unmarked descendants into the list. The tracing threads need to coordinate the selection of an object in the list, they need to synchronize the insertion of objects into the list, and in order to get well balanced work distribution they need to make sure that the work is evenly split.

To measure the scalability of a given heap shape, we imagine an idealized tracing procedure that ignores all of these issues. We first assume that the load balancing is perfect. 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. Second, we assume that within a single clock tick[1], a tracer thread can atomically pick an object A from this 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. Third, we assume uniform memory access times, with no delays due to cache misses, cache line conflicts or false sharing. And last, we assume the 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 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 $P-1$ tracing threads are idle. The fraction of utilized CPU time during the trace is $\frac{1}{P}$ and this fraction is monotonically decreasing with P.

To measure the relevant properties of the heap shape, we have modified Jikes RVM collector to measure 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, the first P objects are pulled out of the work list and all their descendants are added to the list. Note that because the assumptions above, this is inherent to the heap shape. It is independent of both the collector implementation and the particular

[1] 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.

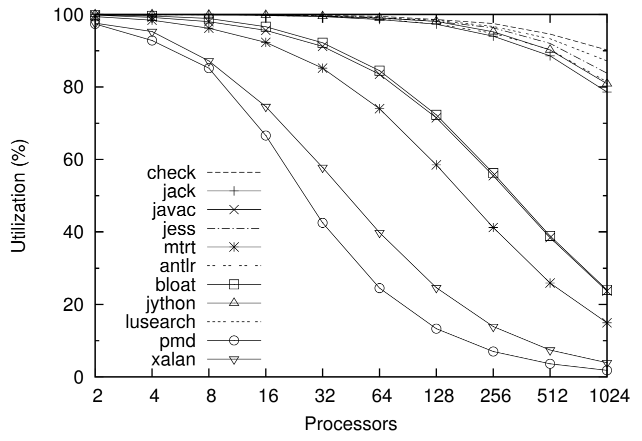

Figure 2. Worst case object graph idealized trace utilization for various levels of parallelism. The lines of *javac* and *bloat* are hard to distinguish as they almost collide, due to similar behavior.

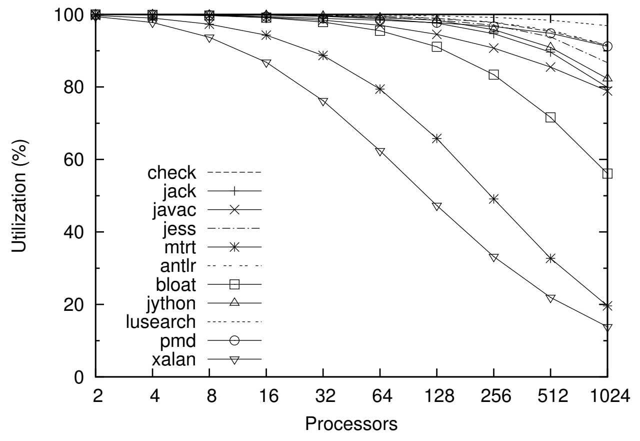

Figure 3. Average object graph idealized trace utilization for various levels of parallelism.

hardware employed. Also, it can be measured on a single processor, i.e., evaluating this measure does not require a many-core platform.

From the above gathered information we distilled a single number, representing the processor utilization, which we report in this paper. For each clock tick, if P objects are available in the work list then all processors are utilized. However, if there are $t < P$ objects available for scanning in the work list, then t processors are utilized at that clock tick, but $P - t$ are not utilized, i.e., are idle. Going over the entire clean trace execution, we can compute the utilization for each clock tick, and also the percent of processor utilization during the entire run. This idealized trace utilization is the measure we propose for the scalability of a given heap shape, and this is what we report next for our benchmarks.

3.1 Idealized Trace Utilization Measurements

We ran a modified version of Jikes RVM, computing the idealized trace utilization measure for $2 \ldots 1024$ processors. 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 idealized trace utilization measure for each GC cycle and then observed the average and minimal values (over all collection cycles) for each benchmark run. Results are presented in Figures 2 and 3, where we show the worst case and average utilization as a function of the number of working processors P for every benchmark.

Recall that the 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 machines with up to eight processors, which are typical of many of today's parallel platforms, the average-case scalability of all benchmarks looks good. Even on the worst-case measurements, some benchmarks exhibit 15% idle time, which is still reasonable. At the level of 32 processors, which is 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*.

4. Improving the Scalability of Object Graph Tracing

The above measurements revealed the existence of Java benchmarks that generate heaps with inherently non-scalable shapes, limiting the scalability of parallel tracing. Poorly-scalable heap shapes should be avoided, since they may prevent Java applications from scaling on multiple computation cores. We proceed with proposing and studying possible solutions.

The main idea is to have runtime systems attempt to handle non-scalable structures and avoid problems that can result from certain patterns in them. There are various ways to handle this problem. Here we present our investigation of two approaches where problematic object-graph patterns are handled by the garbage collector.

4.1 Adding Shortcuts

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. The goal is to shorten the depth of objects, by creating shortcuts into long narrow structures. This yields a more shallow graph, which in turn, yields more work for the tracers to execute at earlier stages of the trace.

In this work, we built a prototype and used it for assessing the shortcuts method on standard benchmarks. Our prototype runs the benchmarks, and after each normal garbage collection trace, the application is paused. During that pause, the prototype fixes the heap using shortcuts as described below and then runs the trace again to observe how much the utilization has improved. Our scheme introduces one additional reference slot in the header of each object in the heap, initiating it with Null. When a potential shortcut to a deep object is discovered, the shortcut edge is installed in the header for use by the collector. We next describe how shortcuts are chosen to be added to the heap. Measurements and results are presented in Section 5.1.

Consider a heap object O and its 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 should, we need to select an object in O's subgraph as the target. To decide whether a 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 *size*, and if the ratio of the depth to the size of the subgraph is larger

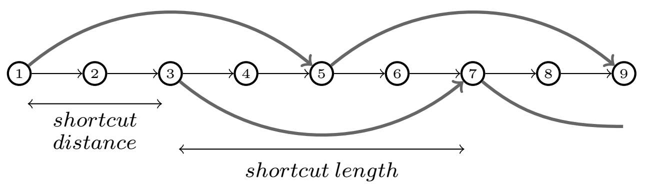

Figure 4. An example of a linked list with shortcuts added. Here, the *shortcut length* is 4 and the *shortcut distance* is 2.

than *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 parameters, we add a 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 *shortcut length*, which is always set as the length of all shortcuts. Also, 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 also maintain a distance between installations of shortcuts, which is also a parameter denoted *shortcut distance*. An example is depicted in Figure 4.

We devised and implemented an algorithm that adds shortcuts to the graph during a traversal. The algorithm is a modified DFS traversal that upon retreating to a parent O during the traversal, retains enough information from the lower nodes to be able to evaluate the size and depth of O's subgraph. Furthermore, if a shortcut is required, then this modified DFS algorithm provides a target object in the subgraph whose depth is *shortcut-length* with respect to O. At this point, the tracer installs a shortcut from O to the target. Our modified DFS was incorporated into Jikes RVM and the implementation was used to check the efficacy of the shortcuts. We do not elaborate on this algorithm in this short version of the paper, as it is not a target in this investigation, and its description requires more space.

To obtain the measurements presented in Section 5.1, we computed the idealized trace utilization measure, described in Section 3 above, before and after the shortcuts were added. We removed all the shortcut references after each collection so they have no bearing on the next collection cycle. The algorithm can be run by several tracers simultaneously with no additional synchronization. Note that this scheme cannot fail the correctness of the garbage collector as no object can cease to be reachable as a result of adding shortcuts.

Our implementation is a prototype because we have conveniently added shortcuts to the heap while the world is stopped. In a real implementation one has to decide when to add the shortcuts and how to maintain them efficiently and effectively.

4.2 Tracing Randomly in Parallel

The second approach we investigated does not modify the heap shape at all but attempts to use idle processors to trace random elements in the heap. The idea is that when a deep data structure actually exists in the heap, then the trace cannot be well parallelized and processors become idle during this time. Such idle threads can be used to trace objects randomly and when lucky, aid the tracing effort and increase scalability. This method was mentioned in [12] but as far as we know has not been investigated prior to this work.

Denote the effort of the normal tracing threads as the *main trace*. While the main trace is executing, a thread that becomes idle selects 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.

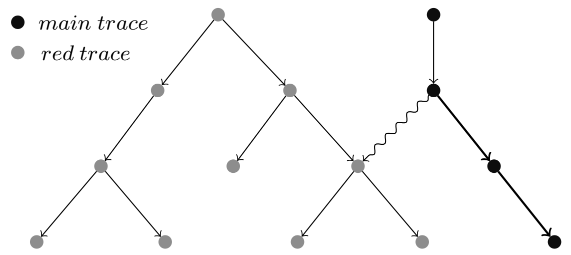

Figure 5. An example of a red trace, which is hit by the main trace. Note that only some of the red vertices are reachable. The rest create floating garbage.

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. It is possible that the main trace discovered a reachable red object, but one that is deep in the trace of the idle thread. All its predecessors in this trace may actually be unreachable. An example is depicted in Figure 5. 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 too. This creates an inaccuracy in the trace and implies floating garbage [2]. To restrict the amount of floating garbage, we limit the number of objects that an idle thread marks with any single color by the *trace-limit* parameter. Second, note that there is a need to synchronize access to the color marks in the objects and to detect trace completion correctly. This synchronization is essential to achieve algorithmic correctness and can add an overhead.

In the implementation we investigated, each idle thread chose a random unmarked element, obtained a unique color and marked the object and its descendants by the unique colors, until *trace-limit* objects were marked. At this point, it recorded the objects on its trace list, i.e., the objects that had been marked by the unique color, but whose children had not yet been visited, in a special list associated with this color. It then looked for more tracing work to be executed. When the main trace encountered an object marked by a color c, it made a note that c is reachable and added the objects from the c-trace list to the main trace list. A different encounter is possible when a trace of color c_1 traces into an object that has already been marked by a different color c_2. In this case, we make a special note that the reachability of c_1 implies the reachability of c_2. In general, we keep such records for each pair of colors. Thus, when a color is discovered to be reachable, all colors reachable from it are immediately identified as reachable as well. The trace is completed when all the main tracers are done and trace lists of all the reachable colors are processed. Note that processing a reachable color's trace list can make more colors reachable so that their lists have to be processed as well.

Our prototype implementation does not really run on a many-core but on an eight-core machine. We decided to check this method, by letting four cores run the main trace, while the other four cores ran the task of the idle threads, picking objects at random

[2] *Floating garbage* is a term that denotes all unreachable objects that the collector does not identify as such, and does not reclaim.

and tracing their descendants. This execution gave us an indication of how well this method works when half of the processors are idle and perform random traces during the execution of the main trace. It simulates a scenario in which half of the threads make progress in the main scan, and at the same time, half of the threads seek random traces to aid the main trace.

As before, a real implementation would have to make various implementation decisions that we settled in ways convenient to us and suitable to the system with which we worked. First, we chose the objects to be scanned at random in the following manner. Each thread has a random number generator seeded with a unique seed. When a new root is needed, the thread generates two new random numbers. The first random number is used as a heap offset to obtain the beginning of an allocation block. We then inspected objects in this block, using the second random number to skip slots between the inspected objects. Objects that were found to be already marked by the regular or the additional tracers were excluded from the search, as were primitive arrays that contain no pointers.

The parameter that needs tuning is the *trace-limit*, which limits the number of objects that can be traced with each unique color as discussed above.

4.2.1 Random choices with filters and biases

In the simple scheme, unmarked objects are chosen randomly in the heap, and their descendants are traced. Note 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.

Our initial measurements were 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 measurements. 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 is 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 recognize all the possible deep structures. It must be remembered that this filtering test is only a prototype meant to validate the potential of further optimizations.

5. Results

We 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 ran the prototypes on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors.

As discussed above, we did not measure the execution time of the prototypes, as the above platform is not a many-core machine. We evaluated the idealized trace utilization measure on the different heap shapes. Therefore, 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 nondeterminism resulting from picking roots at random.

5.1 Adding Shortcuts

In this section we report the results of computing the idealized trace utilization measure before and after adding shortcuts as explained in Section 4.1. 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 Section 4.1 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.

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 3.1, 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 3.1 for up to 512 processors. However, as can be seen in Figures 2 and 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 6 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 and 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 7. 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 8. 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 9. 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 our default set of parameters. It was noted before (see Table 1) that these two benchmarks generate deep object-

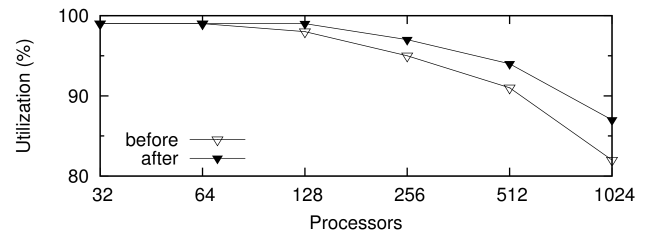

Figure 6. Worst case object-graph trace utilization before and after adding shortcuts for *jython*.

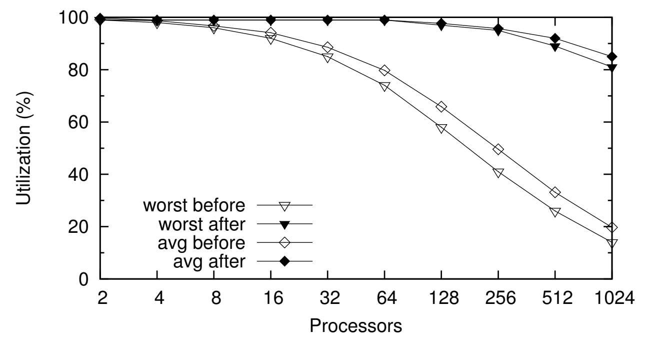

Figure 7. Worst case and average object-graph trace utilization before and after adding shortcuts for *mtrt*.

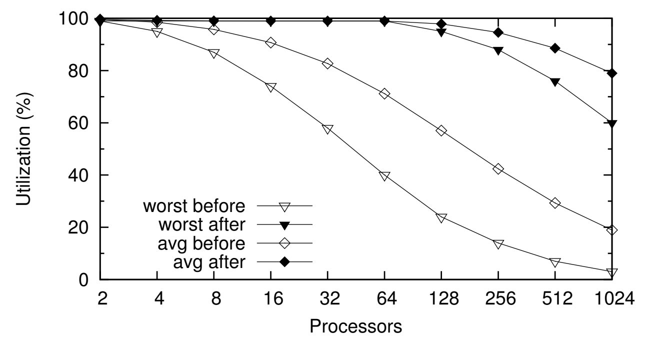

Figure 8. Worst case and average object-graph trace utilization before and after adding shortcuts for *xalan*.

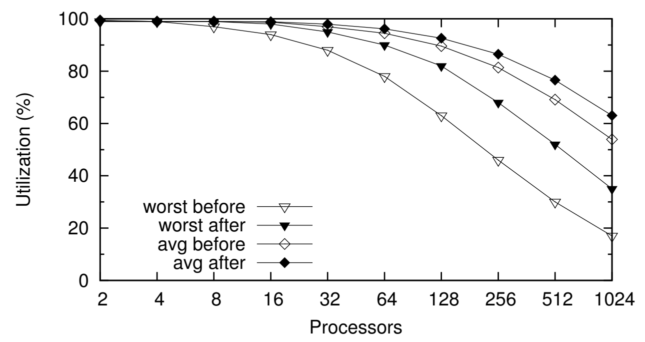

Figure 9. Worst case and average object-graph trace utilization before and after adding shortcuts for *bloat*.

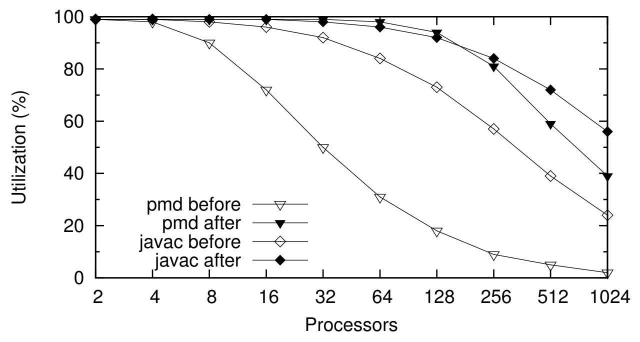

Figure 10. Worst case object-graph trace utilization before and after adding shortcuts for *javac* and *pmd*.

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 10 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 10.

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 Sec-

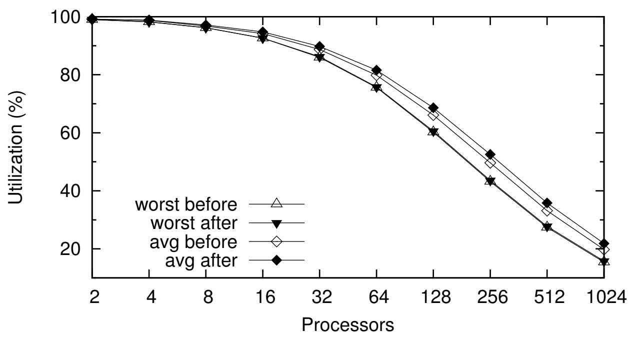

Figure 11. The change in utilization when using random trace without the filter for *mtrt*.

tion 3 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 some connected subgraphs are colored in various colors. Some of these colored components are considered reachable and some unreachable, depending on the reachability status of the 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 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 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 3. 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 4.2. 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 11, 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 4.2.1, the improvements become significant, see Figure 12. 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 13), 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, but the improvement was negligible as shown in Figure 14.

To understand the behavior of the random tracing algorithm, we report some statistics collected during the trace in Tables 2 and 3. 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,

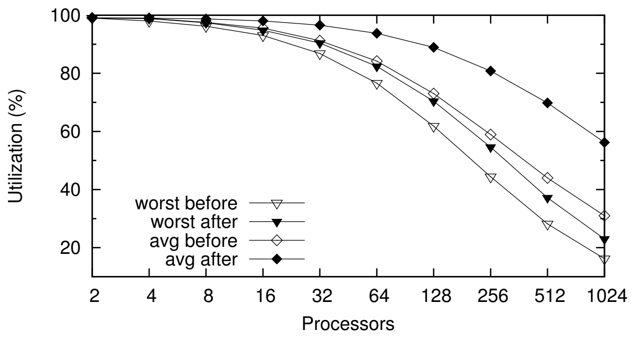

Figure 12. The change in utilization when using random trace for *mtrt*, with the filter described in Section 4.2.1.

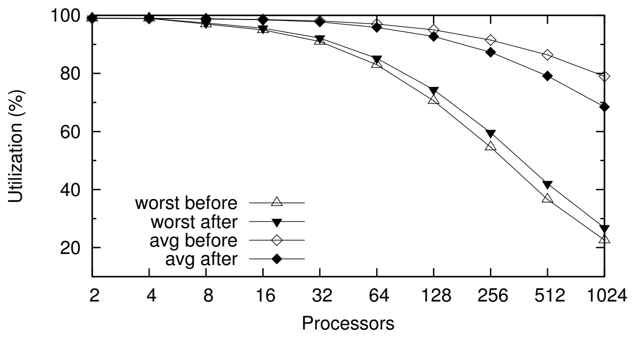

Figure 13. The change in utilization when using random trace without the filter for *javac*. While there were a lot of GC cycles with improved utilization, including the worst case, the average utilization became worse due to many dead recursive structures traced by the special trace in other GC cycles.

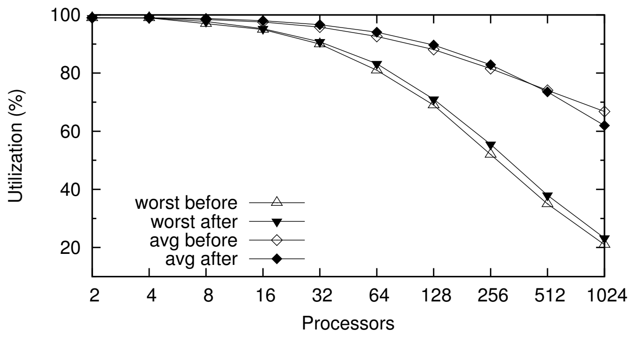

Figure 14. The change in utilization when using random trace for *javac*, with the filter described in Section 4.2.1.

name	live objects (thousands)	useful helpers work (%)	wasted helpers work (%)	floating garbage (%)
db	384	3.99	0.01	0.01
jack	110	11.7	0.16	0.072
javac	316	9.3	0.16	357.6
jess	128	11.9	0.17	0.12
mtrt	363	8.69	0.72	12.52
antlr	220	9.9	0.95	7.14
bloat	380	7.94	0.05	0.07
hsqldb	3,360	4.11	0	0
jython	128	3.31	0.04	0.023
lusearch	230	9.16	0.08	0.17
pmd	370	7.25	0.41	37.24
xalan	330	5.89	0.07	0.84

Table 2. Properties of the random trace with a trace-limit of 1000 (no filter): 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. All the percentages are of the overall number of live objects.

name	live objects (thousands)	useful helpers work (%)	wasted helpers work (%)	floating garbage (%)
db	384	10.8	0.036	0.004
jack	110	25.8	0.08	0.024
javac	316	14.61	0.14	27.01
jess	128	30.22	0.21	0.045
mtrt	363	18.25	2.31	0.92
antlr	220	13.93	1.35	0.13
bloat	380	17.58	0.13	0.2
hsqldb	3,30	4.8	0.023	0
jython	128	12.2	0.1	0.03
lusearch	230	20.75	0.78	0.467
pmd	370	14.39	0.58	51.24
xalan	330	15.74	0.11	0.98

Table 3. Properties of the random trace. This table is similar to Table 2 except that the random choices filter described in Section 4.2.1 was used.

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 *useful* because it saves time for the main trace and parallelizes the trace. Tracing objects of type (2) is *wasted*, since it does not help the main trace and does not harm it. Tracing objects of type (3) is *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 *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 2 we report the results obtained without the filter and in Table 3 – with the filter. It can 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.

6. Related Work

Much of the related work was already mentioned when relevant in the paper. We further mention the most relevant such work here.

Researchers developing garbage collection algorithms have noted long ago [9] that deep linked lists of objects can be a source of imbalance and poor scalability for parallel heap tracing. Developers of load balancing algorithms for parallel garbage collection trace [19, 22, 6, 34] 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 object-graph depths was reported by Siebert [30] in the intention of pointing out problems with highly parallel tracing. They reported long linked lists in several SPECjvm98 benchmarks and offered theoretical prediction on how maximal object-graph depth can influence the scalability of garbage collection tracing. Another relevant study trying to predict the scalability of parallel garbage collection tracing was performed back in 2001 by Endo et al [20]. They measured the object-graph depths to make scalability predictions. Ming Wu and Xiao-Feng Li [34] report that they did not encounter the problematic object-graph layouts in their experiments with the *pseudojbb* benchmark, which is a work constrained version of the *SPECjbb* benchmark. This is consistent with our result that *SPECjbb* has a very consistent shallow object-graph of depth 24.

We improve over the above previous work by extending the benchmark coverage. We also propose and study the idealized trace utilization measure, which provides more information on the scalability of a heap shape. The measurements in this work are consistent with the partial results obtained in previous work and provide more information on more benchmarks. In the second part of this paper, we also investigate avenues to ameliorate the tracing scalability problem. As far as we know, this is the first study of possible ways to ameliorate the heap shape scalability problem.

Click [12] 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.

7. Conclusion

As garbage collected languages remain highly desirable and as the amount of parallelism is steadily rising, expected to reach tens and hundreds of processors available to trace a heap, the question of tracing scalability becomes acute. In this paper we provided an investigation of the heap-tracing scalability. We started by proposing the idealized trace utilization measure. Next, we demonstrated that heap shapes with low scalability are produced by some of the standard Java benchmarks. 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. Further investigation is required to validate the use of these methods within a full garbage-collected system.

Acknowledgments

This paper was initiated in an invaluable discussion with Bill Pugh, that has set the path for this study.

References

[1] Spec: The standard performance evaluation corporation. `http://www.spec.org`.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, Perry Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), February 2000.

[3] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In *OOPSLA, 2003*.

[4] David F. Bacon, Clement R. Attanasio, Han Bok Lee, V. T. Rajan, and Stephen E. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI 2001 [27], pages 92–103.

[5] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), New Orleans, LA, USA, January 2003.

[6] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, November 2005.

[7] Stephen Blackburn, Robin Garner, Kathryn S. McKinley, Amer Diwan, Samuel Z. Guyer, Antony Hosking, J. Eliot B. Moss, and Darko Stefanović. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA, 2006*.

[8] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Edinburgh, May 2004.

[9] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[10] Craig Chambers and Antony L. Hosking, editors. *Proceedings of the Second International Symposium on Memory Management*, ACM SIGPLAN Notices 36(1), Minneapolis, MN, October 2000.

[11] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In PLDI 2001 [27], pages 125–136.

[12] Cliff Click. Private communication.

[13] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the First ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 46–56, Chicago, IL, USA, June 2005.

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

[15] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, USA, January 1994. ACM Press.

[16] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, USA, January 1993. ACM Press.

[17] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Chambers and Hosking [10], pages 155–166.

[18] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *PLDI, 2000*.

[19] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[20] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 43, Washington, DC, USA, 2001. IEEE Computer Society.

[21] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Reducing pause time of conservative collectors. In Hans-J. Boehm and David Detlefs, editors, *Proceedings of the Third International Symposium on Memory Management (June, 2002)*, ACM SIGPLAN Notices 38(2 supplement), pages 12–24, Berlin, Germany, February 2003.

[22] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the First Java Virtual Machine Research and Technology Symposium*, Monterey, CA, USA, April 2001. USENIX.

[23] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.

[24] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[25] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA, 2001*.

[26] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI, 2008*.

[27] *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 36(5), Snowbird, UT, USA, June 2001.

[28] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Chambers and Hosking [10], pages 143–154.

[29] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *OOPSLA, 2002*.

[30] Fridtjof Siebert. Limits of parallel marking collection. In Richard Jones and Steve Blackburn, editors, *Proceedings of the Seventh International Symposium on Memory Management*, pages 21–29, Tucson, AZ, USA, June 2008. ACM Press.

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

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

[33] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`, March 2005.

[34] Ming Wu and Xiao-Feng Li. Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. In *IEEE International Parallel and Distribution Processing Symposium (IPDPS) 2007*, Long Beach, CA, March 2007.